Hypothesis-based concept assignment to support software maintenance

Gold, Nicolas Edwin

Hypothesis-Based Concept Assignment to Support Software Maintenance

Nicolas Edwin Gold

Ph.D. Thesis

Research Institute in Software Evolution
Department of Computer Science
University of Durham

2000

Abstract

Software comprehension is one of the most expensive activities in software maintenance and many tools have been developed to help the maintainer reduce the time and cost of the task. Of the numerous tools and methods available, one group has received relatively little attention: those using plausible reasoning to address the concept assignment problem. This problem is defined as *the process of assigning descriptive terms to their implementation in source code, the terms being nominated by a user and usually relating to computational intent*. It has two major research issues:

- *Segmentation*: finding the location and extent of concepts in the source code.
- *Concept Binding*: determining which concepts are implemented at these locations.

This thesis presents a new concept assignment method: Hypothesis-Based Concept Assignment (HB-CA). A framework for the activity of software comprehension is defined using elements of psychological theory and software tools. In this context, HB-CA is presented as a successful concept assignment method for COBOL II, employing a simple knowledge base (the library) to model concepts, source code indicators, and inter-concept relationships. The library and source code are used to generate hypotheses on which segmentation and concept binding are performed.

A two-part evaluation is presented using a prototype implementation of HB-CA. The first part shows that HB-CA has linear computational growth in the length of program under analysis. Other characteristics addressed include HB-CA's scalability, its applicability to other languages, the contribution made by different information sources, domain independence, representational power, and guidelines for the content of the library. The first part concludes by comparing the method and implementation to cognitive requirements for software comprehension tools. The second part considers applications of HB-CA in software maintenance. Five areas for potential cost reduction are identified: business-rule ripple analysis, code ripple analysis, module selection, software reuse, and software module comprehension.

Acknowledgements

I would like to thank everyone who has helped me with my research. In particular, I am very grateful to my supervisor, Professor Keith Bennett, whose advice, insight, and encouragement have been invaluable throughout. The SABA team provided many hours of interesting discussion and I would like to thank Dr. Magnus Ramage for helping me to explore some of my early ideas. My thanks also go to all the other people in the department who have discussed my work with me. I am grateful to Computer Sciences Corporation (CSC) for providing example source code, and would like to thank Shaun Hexter for his help in particular.

I have thoroughly enjoyed my time as a Ph.D. student and much of this is due to the people I have shared an office with over the years. My thanks go to Antony Hofton, Stephen Rank, Phyo Kyaw, and Claire Knight for many enjoyable discussions, games, and adventures in musical taste(!).

I would like to thank my parents, Linda and Kenneth, and the rest of my family who have provided encouragement, love, and support throughout. They have listened to many explanations of my ideas and have always been happy to read my work. I am grateful to my parents and my in-laws, Sheila, Richard, and Graeme, for proofreading my thesis.

Finally, my greatest thanks go to my wife Helen. Without her unfailing love, encouragement, and confidence in me, the ups and downs of research would have been a harder ride. Life wouldn't be as much fun without her and the support she has given me during this work has been immeasurable. This thesis is dedicated to her with all my love.

This work was funded by EPSRC as part of the Software As a Business Asset (SABA) project in the Systems Engineering for Business Process Change (SEBPC) programme.

Statement of Copyright

Declaration

The material presented in this thesis is the sole work of the author and has not been previously submitted for a degree at this or any other university.

Contents

Chapter 1 Introduction

Chapter 2 Background and Framework

Chapter 3 Hypothesis-Based Concept Assignment

Chapter 4 Hypothesis Generation

Chapter 9 Evaluation II: Applications of HB-CA

Chapter 10 Conclusions

Appendix Investigation Data

References

Figures

Tables

Chapter 1
Introduction

1.1 Context

Software maintenance is an important part of the software lifecycle, typically accounting for at least 50 percent of the total lifetime cost of a software system [LIEN80]. Consequently, it is desirable to reduce the cost of software maintenance whilst preserving the quality of the software system and maintenance process.

The state of a software maintenance process can be assessed with methods such as the Capability Maturity Model (CMM) [PAUL93]. A reasonably mature process (e.g. CMM Level 3 or higher) will have a number of distinct phases; the IEEE standard for software maintenance [IEEE98] defines seven:

a) Problem/modification identification, classification, and prioritisation;
b) Analysis;
c) Design;
d) Implementation;
e) Regression/system testing;
f) Acceptance testing;
g) Delivery.

Reducing the total cost of software maintenance requires the individual cost of one or more of the constituent phases to be lowered.

1.2 Area of Interest

Many authors have acknowledged the central role and high cost of software comprehension within software maintenance, either directly (e.g. [MAYR97], [STAN84]), or indirectly, as a consequence of software complexity (e.g. [BANK93]). Estimates of the time spent performing this activity vary. Hall claims that understanding the documentation and logic of programs occupies 47-62 percent of

maintenance programmers' time [HALL87a] [HALL87b]. Parikh and Zvegintzov suggest that more than half the programmer's task is in understanding the system [PARI83], and Standish claims that it may be the dominant cost in the entire software lifecycle [STAN84].

Software comprehension takes place in several phases of the maintenance process described in section 1.1, although the IEEE standard does not make this explicit (see [IEEE98]). It is primarily undertaken during design and implementation where modules are to be redesigned or changed. It could be argued that identifying ripple effects during the analysis phase also requires some understanding of the software modules. Software comprehension is an ideal starting point for reducing the overall cost of software maintenance because of its importance, high cost, and frequent occurrence in the maintenance process.

A common approach to reducing the cost of the maintenance process is the provision of automated assistance to software maintainers. The task to be performed and the expertise of a particular maintainer determine the type of tool that is appropriate in a given situation. Novice and expert maintainers understand code in different ways. Novices tend to take a syntactic approach to understanding a program, organising their knowledge structures around the program syntax. Experts organise their knowledge around algorithms and functional characteristics within their domain of expertise [MAYR95]. The work presented in this thesis is aimed at assisting expert maintainers with software comprehension. Consequently, the focus is on tools that automatically identify the implementation of algorithms, abstractions, and domain concepts in software. Tilley and Smith claim maintainers most lack such tools [TILL95] and evidence that higher-level semantic knowledge reduces maintenance effort [RAMA96] strengthens their case.

There are many types of software tool available to help with software comprehension, emphasising different aspects of software systems and modules, and usually creating new representations for them. Biggerstaff et al. differentiate between naïve and intelligent agents (tools) for providing such representations [BIGG93]. Naïve agents generally perform deductive or algorithmic analysis of program properties or structure, e.g. program slicers (see [TIP94]) or dominance

tree analysers (see [BURD99]). Intelligent agents attempt to assign descriptions of computational intent to source code. Agents in the latter category meet the demand (discussed in the previous paragraph) for tools that can identify algorithms, abstractions, and domain concepts in software.

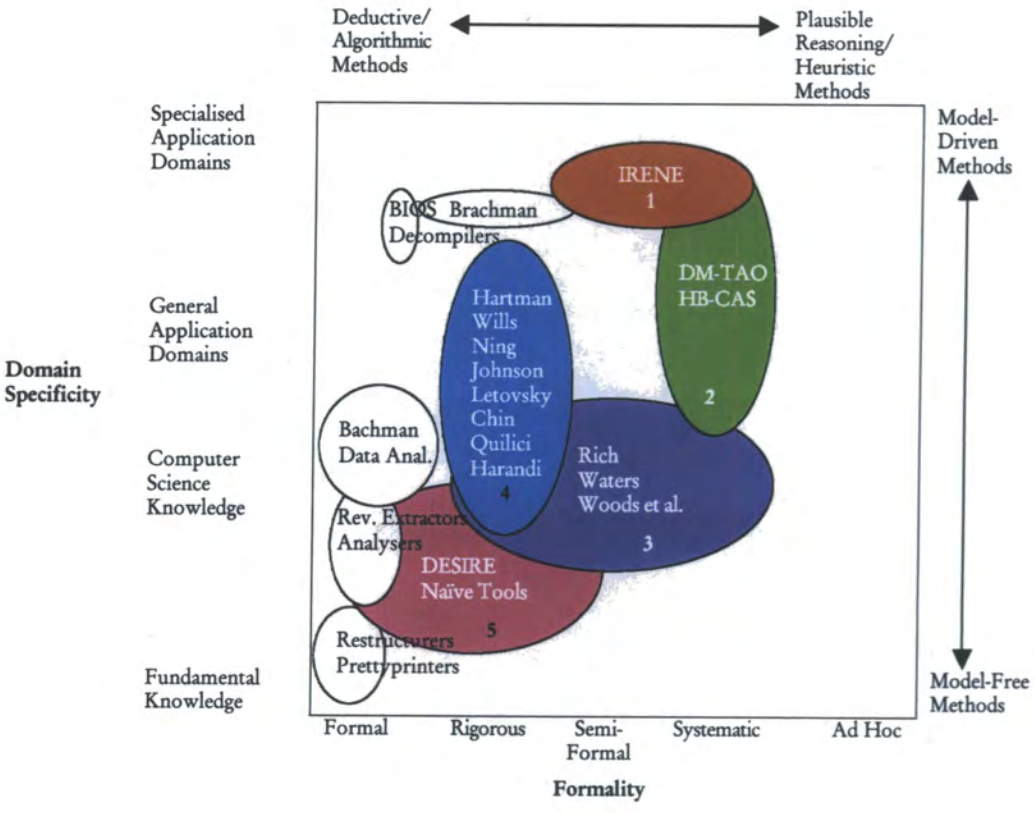
In [BIGG93], Biggerstaff et al. claim that research on intelligent agents can be divided into 3 distinct approaches:

1) Highly domain specific, model driven, rule-based question answering systems that depend on a manually populated database describing the software system. This approach is typified by the Lassie system [DEVA91].
2) Plan driven, algorithmic program understanders or recognisers. Two examples of this type are the Programmer's Apprentice [RICH90], and GRASPR [WILL92].
3) Model driven, plausible reasoning understanders. Examples of this type include DM-TAO [BIGG93], [BIGG94], IRENE [KARA92], and the method presented in this thesis (termed HB-CAS in Figure 1).

One exception to this categorisation is Hartman's work [HART91a] that falls between approaches 2 and 3.

Systems using approaches 1 and 2 are good at completely deriving concepts within small-scale programs but cannot deal with large-scale programs due to overwhelming computational growth. Approach 3 systems can easily handle large-scale programs since their computational growth appears to be linear in the length of the program under analysis. They suffer from approximate and imprecise results [BIGG93].

Figure 1 is based on the summary of the program understanding landscape in [BIGG93]. The original has been updated to include additional work on program understanding, with the number and colour of each oval providing a key to the citations below. Biggerstaff et al. do not refer to publications in the original figure but Figure 1 adds this information.



Key to citations

Oval	Author(s)	System	Citation(s)	Oval	Author(s)	System	Citation(s)
1	Karakostas	IRENE	[KARA92]	4	Wills	GRASPR	[RICH90], [WILL90], [WILL92], [WILL93]
2	Biggerstaff et al.	DM-TAO	[BIGG93], [BIGG94]	4	Ning Kozaczynski	Concept Recogniser	[KOZA94]
2	Gold	HB-CAS	Ph.D. Thesis	4	Johnson	PROUST	[JOHN85], [JOHN86]
3	Rich, Waters	Programmer's Apprentice	[WATE79], [WATE82], [RICH88], [RICH90], [RICH92], [RICH93],	4	Chin, Quilici	DECODE	[CHIN95]
3	Woods et al.	PU-CSP	[QUIL96], [WOOD96a], [WOOD96b], [QUIL97], [ZHAN97], [QUIL98], [WOOD98a], [WOOD98b]	4	Harandi Ning	PAT	[HARA90]
4	Hartman	UNPROG	[HART91a], [HART91b], [HART92]	5	Biggerstaff et al.	DESIRE	[BIGG89], [BIGG93], [BIGG94]

Figure 1: The Program Understanding Landscape

The method presented in this thesis is intended to operate with real-world, large-scale programs and consequently adopts a plausible reasoning approach to its intelligent analysis. Two systems in Figure 1 share this approach: DM-TAO, and IRENE. A brief description of each is given here and fuller explanations of their approaches are presented throughout the thesis.

1.2.1 DM-TAO

DM-TAO forms the intelligent reasoning component of the DESIRE toolkit described in [BIGG89], [BIGG93], and [BIGG94]. It aims to identify domain concepts in C source code, using a connectionist inference engine to determine the appropriate concept for a section of code. A rich domain model embodies a large number of weighted relationships and concept types. Relationship weights are updated automatically based on the actions of the maintainer using the system. DM-TAO can operate in three modes:

1) Conceptual *grep*: search the source code for a user-specified concept.
2) Conceptual highlights: search the source code for any recognisable concept.
3) Identification: suggest a concept for selected code.

Modes 1 and 3 require user involvement in the concept assignment process.

The evaluation of DM-TAO described in [BIGG93] is based on three files (about 600 lines of code) containing data definitions in the domain of multi-tasking windows systems. A manual analysis of the files was undertaken to find the most important concepts for understanding the data. Twenty-seven concepts were found and a domain model constructed containing twenty of them. DM-TAO was tested in conceptual *grep* mode finding twenty of the twenty-seven concepts and producing three false positives, which were attributed to the fact that the connectionist network was weakly trained. In identification mode DM-TAO tended to over-generalise, finding both the appropriate super- and sub-concept for a segment of code. This was attributed to some feature extractors not being implemented, e.g. syntax categories.

This evaluation indicates that DM-TAO is reasonably successful at concept recognition for data definitions. The strongest disadvantage of the approach is the size, complexity, and computational cost of updating the domain model. The method presented in this thesis aims to achieve concept recognition with a considerably simpler and smaller domain model. It is intended to find operational concepts rather than the data declarations on which DM-TAO has been evaluated.

1.2.2 IRENE

The IRENE system employs concept acquisition techniques to retrieve business knowledge from COBOL programs [KARA92]. It embodies a top-down approach, working from a domain-engineered model of business entities to their source code implementations. Relationships between the entities are expressed as dependencies and derivations. The process of concept acquisition (similar to concept assignment) begins with user-supplied hypotheses about the correspondence of certain domain concepts to constructs in the program. IRENE generates further hypotheses using this information. The process is interactive, with the system user verifying concept assignments and assisting with hypothesis generation.

IRENE has been evaluated on a small payroll application of about 500 lines, written in COBOL 74. The internal representation (a parse tree represented as a hierarchy of frames, see [KARA92]) was validated manually but no indication is given in [KARA92] as to the success of the approach.

Since the available literature does not show IRENE's concept assignment/acquisition ability, comparative evaluation is difficult. The top-down approach adopted is the opposite of that used by the method presented in this thesis. In addition, IRENE's use of a moderately rich domain model suffers similar problems of maintenance and high initial cost that affect DM-TAO.

1.2.3 Summary

Although IRENE and DM-TAO adopt different approaches to concept assignment, both systems use complex domain models requiring a large amount of effort to create and maintain. Neither system has been evaluated extensively or on particularly large programs.

1.3 Discussion of Problem

1.3.1 The Concept Assignment Problem

To meet the need for tools that identify algorithms, abstractions, and domain concepts in programs, this thesis addresses the *concept assignment problem*. The term was introduced by Biggerstaff et al. to describe the problem of assigning terms regarding computational intent to appropriate regions of source code [BIGG93]. The emphasis of the work presented here is on *automatic* concept assignment with minimal user involvement, although the activity can also be performed semi-automatically or manually. The latter approaches are likely to incur greater cost.

Biggerstaff et al. define the concept assignment problem as:

> "... a process of recognising concepts within a computer program and building up an "understanding" of the program by relating recognised concepts to portions of the program, its operational context and to one other." [BIGG93]

They refer to two distinct types of concept: programming-oriented, and human-oriented. The former can be detected with traditional parsing technology using formal, structure-oriented patterns of features as signatures for concepts. The term "human-oriented" is used to refer to an informal expression of computational intent e.g. acquire target. The signature for such concepts (also termed domain concepts in this thesis) is less well defined and open to variation. The model of concept recognition required for domain concepts is characterised as an opportunistic, non-deterministic, and chaotic piecing together of evidence for a concept, until some threshold of confidence is reached about its identity. This contrasts with the programming-oriented model of recursive, algorithmic, deterministic, and orderly building of abstract components from less abstract components [BIGG93].

A domain is defined as a problem area [DEBA94] but it is an overburdened term [TILL96a] and as such, it is often difficult to define the limits and contents of any one in particular. Using terms such as "programming-oriented" and "domain-oriented" to differentiate types of concept may be ambiguous in some circumstances, e.g. programming-oriented concepts are concepts in the domain of programming and hence are domain concepts. In order to avoid this confusion,

and to define more precisely the problem addressed by this work, the concept assignment problem can be rewritten as:

> "The process of assigning descriptive terms to their implementation in source code, the terms being nominated by a user and usually relating to computational intent."

This problem statement captures much of the original definition while removing ambiguity from the supporting terms. A concept is regarded therefore as a descriptive term nominated by the user. The rewritten problem statement is concerned solely with the essence of concept assignment, i.e. mapping concepts to code. Relating these concepts to the operational context of the program and to each other is not within its scope.

1.3.2 Research Issues

Tilley et al. state that concept assignment research is at a very early stage, partly due to the complexity of the matching process [TILL98b]. Two major research issues can be identified within the overall concept assignment problem:

- *Segmentation*: finding the location and extent of concepts in the source code.
- *Concept Binding*: determining which concepts are implemented at these locations.

Segmenting a program involves grouping pieces of conceptual information generated from the source code. Concept binding involves analysing these groups for the most plausible concept assignment for each.

1.3.3 Problem Boundaries

The concept assignment method presented in this thesis has been developed with the assumption of certain problem boundaries and applications.

Globally, organisations maintain a large amount of COBOL and this provides a strong motivation for targeting the technique at this language and its variants. Analysis is targeted therefore at programs written in IBM COBOL II. In view of the aim of determining computational intent, the problem is restricted to the

procedure division of such programs and functional concepts are considered more important than data concepts.

The objective is to support a single maintainer in software comprehension, and consequently solutions are not expected to support group-based comprehension. In addition, it is assumed that such solutions will operate on individual modules of code.

In summary, this thesis addresses the concept assignment problem for the procedure division of programs written in IBM COBOL II. The problem is restricted to analysing one module at a time, presenting the results to a single maintainer.

1.4 Research Aims and Criteria for Success

The aims of this research, and hence the criteria for success, cover many aspects of both the problem and solution. A framework to model the comprehension activity for people and software tools is required to enable comparative evaluation of cost later in the thesis. The concept assignment problem must be solved and a prototype tool constructed to demonstrate the viability of the solution. The criteria for success are formally stated thus:

1) The definition of a framework for the activity of software comprehension. This should capture the essential processes and data structures involved in software comprehension, regardless of whether the actor (i.e. the entity undertaking the comprehension activity) is a person or a software tool.

2) The creation of a formal model of the comprehension activity framework discussed in criterion 1 to define clearly its data structures.

3) The development of a new method to undertake automatic concept assignment using a simple knowledge base. It should be capable of analysing real-world COBOL II code and successfully cope with poorly structured and monolithic programs, in addition to well-structured

examples. The method should provide a software maintainer with automatically recognised concepts linked to regions of source code.

4) As part of criterion 3, the development of novel approaches to address the two main research issues in concept assignment: segmentation, and concept binding.

5) The extension of the general formal model (see criterion 2) to the new concept assignment method.

6) The implementation of a prototype tool to demonstrate the feasibility of the new concept assignment solution. This should allow easy evaluation of the method.

Chapter 10 presents a discussion of the success of this research with reference to these criteria.

1.5 Evaluation Criteria

The primary objective of this work is to define a method to perform automatic concept assignment using a simple domain model. It should be capable of handling real-world programs and perform successfully, whatever the structural quality of the code being analysed. Chapters 8 and 9 present an extensive evaluation of the method described in this thesis. The first part (shown in Chapter 8) is based on the following criteria:

- Representational Issues
  - Spatial Cost
  - Representational Power
  - Library Content
- Performance Issues
  - Segmentation
  - Concept Binding
  - Computational Cost
  - Scalability

- General Issues
    - Domain Independence
    - Language Independence
    - Expandability
    - Cognitive Requirements

The three groups of criteria cover a wide range of characteristics. The method is evaluated in Chapter 9 to establish where it may be applied in the software maintenance process.

1.6 Contribution

The main contribution of this work is a new method for automatic concept assignment: Hypothesis-Based Concept Assignment (HB-CA). It uses a simple knowledge base and is targeted at COBOL II. The two main research issues within the concept assignment problem are addressed:

- *Segmentation*: Structural information and self-organising maps are used to cluster related concept hypotheses. This approach allows the method to handle well-structured, poorly-structured, and monolithic code.
- *Concept Binding*: Concept clusters are analysed and scored. Ambiguity is resolved through the application of simple rules.

The method is set in the context of a framework describing the software comprehension activity. This captures the essential data structures and processes of software comprehension for both people and software tools. A formal model of the framework expresses its data structures in set theory. HB-CA is compared to other concept assignment solutions throughout this thesis and an extensive evaluation of the method and its use in the software maintenance process is presented.

1.7 Thesis Structure

This thesis is divided into ten chapters.

Chapter 1 introduces the motivation and context for the research, discusses the problem to be solved, and sets out the research aims and criteria for success.

Chapter 2 develops a framework modelling software comprehension. Parts of this framework are formalised using set theory.

Chapter 3 introduces Hypothesis-Based Concept Assignment. Comparisons are drawn with the methods underlying the DM-TAO and IRENE systems. The framework and formal model presented in Chapter 2 are extended for HB-CA.

Chapter 4 describes the first stage of HB-CA, hypothesis generation, in the context of the comprehension activity framework and formal model. HB-CA's hypothesis generation method is compared with other systems' techniques for gaining initial information about a program.

Chapter 5 presents a method for segmenting programs based on their structure, and using self-organising maps of concept hypotheses. This forms the second stage of HB-CA. Appropriate comparisons are made with other systems' methods for segmentation.

Chapter 6 describes the final part of HB-CA: concept binding. HB-CA's method is compared with those used by DM-TAO and IRENE.

Chapter 7 describes a prototype implementation of HB-CA called the Hypothesis-Based Concept Assignment System (HB-CAS). The implementation is used in the investigations presented in Chapters 8 and 9.

Chapter 8 presents the first part of a detailed evaluation of HB-CA. The criteria outlined in section 1.5 are used to evaluate the method.

Chapter 9 contains the second part of the evaluation, examining applications of HB-CA in the software maintenance process.

Chapter 10 contains a general discussion and summary of the work accomplished. The success of the research is considered in terms of the criteria presented in section 1.4 and ideas for further work are suggested.

The Appendix contains data and results pertaining to the investigations carried out in the evaluation. It is followed by a list of references.

1.8 Summary

Chapter 1 has introduced the work presented in this thesis. The motivation and context of the research have been explained with reference to other achievements in the field. Two major research issues have been identified within the concept assignment problem: segmentation, and concept binding. Evaluation criteria have been presented and the structure of the thesis explained.

Chapter 2 discusses background material and develops a framework to model the software comprehension activity.

Chapter 2
Background and Framework

2.1 Introduction

Chapter 1 introduced the material in this thesis, presenting the context and motivation of the work. The research problem was defined and two key issues identified. Criteria for evaluating both the method and the research were presented. The structure of the thesis also was discussed.

This chapter examines the background to the method presented in this thesis. A standard process of software maintenance is described and issues relating to its improvement are discussed. A descriptive framework capable of modelling both human and automated approaches to software comprehension is then introduced. This is the context for the Hypothesis-Based Concept Assignment (HB-CA) method presented in later chapters. The framework's source and target representations are formally defined.

2.2 Software Maintenance

The IEEE definition of software maintenance given in [IEEE98] is:

> "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

It is the largest and most expensive stage of the software lifecycle [ROBS91] potentially consuming 70 percent of the total lifecycle costs [LIEN80].

2.2.1 Types of Software Maintenance

Swanson describes three types of software maintenance [SWAN76]:

- **Perfective**: Changing or adding to a system's functionality, improving maintainability, enhancing performance.
- **Adaptive**: Changing a system to account for environmental changes.
- **Corrective**: Fixing bugs in a system.

A change to a software system will usually involve elements of these types of maintenance [SOMM93].

2.2.2 The Software Maintenance Process

Organisations undertake the process of software maintenance in various ways ranging from ad hoc and disorganised, to highly controlled and well managed. In recent years, there has been great interest in the improvement of the software maintenance process, with a view to controlling and reducing the cost of the task.

It is possible to conduct the improvement of an organisation's software process in a managed way. One of the best known and most widely used examples of this is the Capability Maturity Model (CMM) developed by the Software Engineering Institute. The CMM defines five levels of software process maturity for an organisation although it does not specifically prescribe how an organisation should move from one to another. The levels are described below [PAUL93]:

- **Level 1: Initial**

  The software process is ad hoc, occasionally chaotic. Few activities are defined and success depends on individual effort. There is little predictability in quality, budget, schedule, or functionality.

- **Level 2: Repeatable**

  Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications. Planning and management of new projects is based on experience with similar projects.

- **Level 3: Defined**

  The software process for both management and engineering activities is documented, standardised, and integrated into a standard software process for an organisation. All projects use a tailored version of this standard process.

- **Level 4: Managed**

  Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

- **Level 5: Optimising**

  Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

Level 1 establishes a baseline against which process improvements in the higher levels can be compared. The activities that an organisation can undertake to establish or improve the software process are characterised in Levels 2-5 [PAUL93].

There is a current initiative to standardise the various approaches to software process improvement and assessment. It aims to reduce the cost of assessing process capability for organisations and their customers by defining certain criteria that must be met by a process assessment method. The results of using differing methods can then be compared within the framework. The initiative is called SPICE and is documented in [EMAM98].

Once an organisation has reached Level 3 of the CMM, the stages of the software maintenance process are defined. The IEEE software maintenance standard defines seven stages for the maintenance process [IEEE98]:

a) Problem/modification identification, classification, and prioritisation;
b) Analysis;
c) Design;
d) Implementation;
e) Regression/system testing;
f) Acceptance testing;
g) Delivery.

Chapter 1 identified the analysis, design, and implementation stages as having particular relevance to the work presented here. The IEEE definition of these stages (see [IEEE98]) is discussed in the next few sections to show where software comprehension is required.

2.2.2.1 Analysis

Analysis is an iterative process that has at least two components: feasibility analysis, and detailed analysis. The modification request, system and project documentation, and repository information are used to determine the feasibility and scope of the modification. Where documentation is inadequate and source code is the only reliable reference for the current system, reverse engineering is recommended. Various activities are required during analysis including the identification of elements involved in the modification, determination of the modification's impact, identification of short and long-term costs, and implementation planning. The standard suggests that a preliminary modification list of those elements affected is created, e.g. software, specifications, database, and documentation. This involves some degree of software comprehension, probably at the system rather than program level, to determine which elements may be affected. It is interesting to note that although analysis requires the identification of the elements involved, identifying the specific software modules affected is left until the design stage. This could make cost estimation extremely difficult in some circumstances.

Identifying the impact of the modification and building the preliminary modification list may involve ripple analysis. Ripple analysis involves assessing the effect of a change on other parts of a system and can be undertaken in various ways. The analysis stage of the software maintenance process is likely to require ripple analysis at the business-rule level primarily, as the identification of affected software modules is not addressed until the design stage.

Business rules have been defined as:

> "A requirement on the condition or manipulation of data expressed in terms of the business enterprise or application domain." [SELF93]

A key idea is that the rule is stated at the level of the application domain, not of programming. Consequently, business rules are related closely to domain models but reflect the desires of a particular company, not the general features of a domain [SELF93]. Examples of business rules might be found in the formulae and conditions that define the growth and charging structure of a financial product such as a pension policy. These make certain requirements of manipulations on the entities involved in the management of the policy.

Ripple analysis in terms of business rules poses the following question: if one rule is changed, are others also affected? Finding affected rules may require examination of documentation and software, with the cost of undertaking such analysis likely to be crudely proportional to the number of artefacts that need to be inspected. Business-rule ripple analysis can be seen as an example of the higher-order impact analysis that Tilley and Smith describe in [TILL96b]. Higher-order impact analysis allows the software engineer to analyse proposed changes at the application-domain level rather than the implementation-domain level [TILL96b].

The analysis phase produces a report that forms part of the input to the design phase.

2.2.2.2 Design

This phase uses the system and project documentation, source code, comments, databases, and the output of the analysis stage to design the modification to the system. The process includes identifying affected software modules, modifying their documentation, creating and identifying test cases, and updating the modification list. The whole phase involves software comprehension but two activities particularly require it: code ripple analysis, and module selection. Both are part of the process of identifying affected software modules.

Code ripple analysis answers a similar question to that posed for business rules above. In this situation however, the ripples are examined on the basis of potential changes to source code. This can be undertaken at a syntactic and semantic level, e.g. using a forward program slice, see [NING94]. Alternatively, it could be conducted conceptually in a similar manner to business-rule ripple analysis, with the difference lying in the type of concept being considered. Code ripple analysis is more likely to be dealing with software engineering concepts than application-domain related concepts.

The cost of code ripple analysis is addressed extensively in the literature but usually in terms of specific algorithms. Since the work presented in this thesis is not concerned with particular methods for the process, the cost can be regarded as roughly proportional to the number and size of the artefacts examined.

Module selection is the process of determining which modules are affected by a proposed change. It can take place before and/or after ripple analysis and involves searching the code repository for instances of concepts or code that are known to require change. The cost can be seen as a function of the size of the code repository (in terms of total lines of source code) and the search method. Translating the behavioural description of a modification to its implemented counterpart can be extremely difficult. Concept-based search could assist with selecting the modules that need changing.

2.2.2.3 Implementation

Implementation involves making the specified changes to the system. The IEEE standard suggests that implementation should be commenced during the design phase, particularly if the change is complex, in order to better understand the modification. The standard defines four sub-processes: coding and unit testing, integration, risk analysis, and test-readiness review. Software comprehension is particularly required in coding and unit testing. Although the standard does not elaborate further on the coding sub-process, it is possible to break it into two parts: software module comprehension, and change implementation [GALL91]. These parts may be iterative. Software module comprehension is the process of studying the software module to be changed, in order to understand where and how the change should be made. Once the module is understood, the change can be made. Achieving such understanding is a non-trivial task accounting for a very high proportion of the total cost of software maintenance. The work presented in this thesis is aimed at helping to reduce understanding cost through automatic concept assignment. The effort of understanding can be seen as proportional to the size of the module being considered, although this relationship may not necessarily be linear since the maintainer may change comprehension strategy for different sizes of program (see [LITT86]). Other factors such as the program complexity, quality of coding, and maintainer's experience may also have an impact. Familiar modules are likely to take less time to comprehend than those not previously addressed.

2.2.2.4 Summary

The IEEE standard presents a good model for the software maintenance process, identifying the major stages within it. The International Standards Organisation also defines a software maintenance process standard (see [ISO99]). This is a three-stage process that is less comprehensive than the IEEE version. A particular problem is that impact analysis is not undertaken until the implementation stage, potentially causing difficulty with cost estimation.

The stages of the IEEE standard where there is a strong requirement of software comprehension have been discussed and cost factors identified in each. Four particular activities have been highlighted: business-rule ripple analysis, code ripple analysis, module selection, and software module comprehension.

2.3 Software Comprehension

To benefit software comprehension, approaches to automatic assistance must have the potential to lower the cost of producing representations used or required by the software maintainer. This section presents a descriptive framework for the activity of software comprehension. It is specialised for the concept assignment problem and unifies automated and manual approaches. The framework is used later in this thesis for a discussion of the cost savings that might be achieved when using automated assistance in software comprehension.

2.3.1 A Comprehension Activity Framework

Initially, it is assumed that a single maintainer is attempting to understand a software module for one of the reasons discussed in section 2.2.2.

Maintainers gain their understanding of how a module performs a task by using and creating various representations of the software to emphasise different characteristics. These representations can show things such as the control and data flow, or the relationship between subroutines in a module, e.g. using a call graph. Source code captures all of these properties and is the most widely used representation. Others are created and used to assist with understanding the source code, and changes to a software module are usually made in the source code first, with other representations updated to reflect these modifications.

The activity of software comprehension can be characterised as the use, creation, and modification of representations of the software by a person. This is shown in Figure 2.

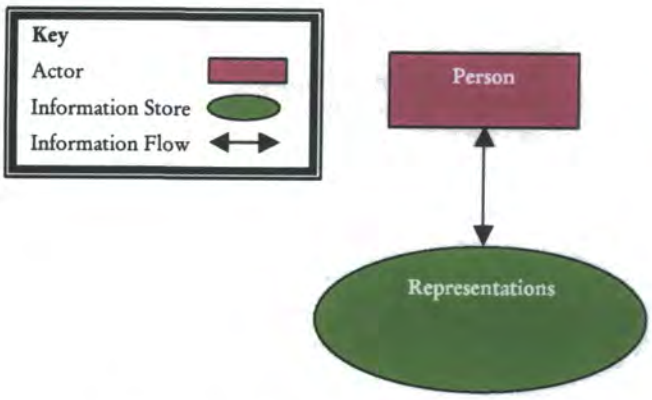


Figure 2: Basic Framework Describing the Software Comprehension Activity

Software tools can be employed to create and use some representations of software, e.g. the object-code representation produced by a compiler. This ability can be captured by generalising the framework shown in Figure 2 to describe a *processor* (which can be a software tool, person, or other device) that can create, modify, and use representations of the software. This is represented in Figure 3.

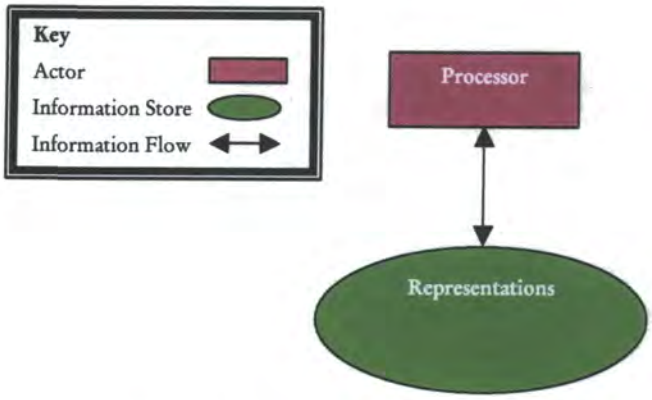


Figure 3: Basic Framework Revised to Describe the Comprehension Activity using a Processor

In many cases the only current, complete, and trustworthy information about a system is its source code; all other information must be derived from this [TILL98a]. Fjeldstad and Hamlen found that when making an enhancement, programmers studied the original program about three and a half times as long as they studied the documentation, and about as long as they spent implementing the change [FJEL79]. Since source code captures many properties of the software and

is usually used as the primary source of information [CORB89] from which other representations are derived, it can be separated justifiably from the other representations in the framework. The framework does not aim to capture the module change activity hence source code is not shown as an output representation.

The framework in Figure 4 captures the source code separately from the other representations.

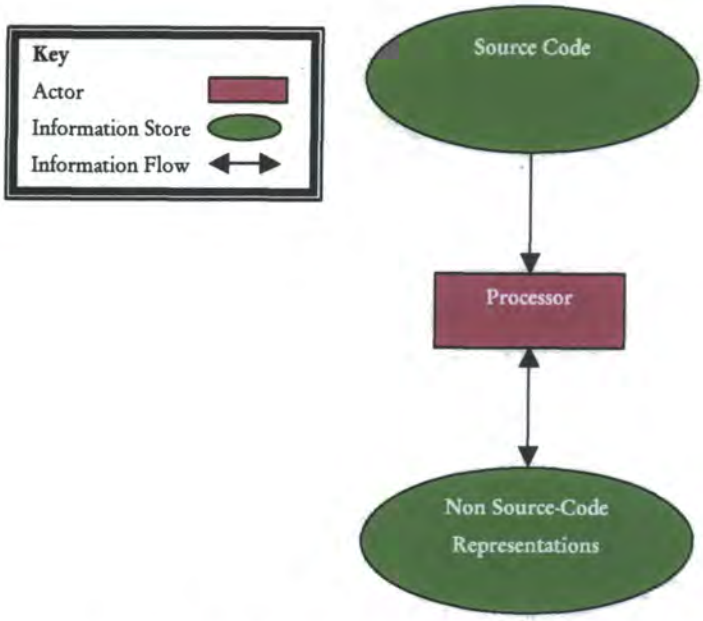


Figure 4: Comprehension Activity Framework Showing Separated Source Code

The description of a general framework for the software comprehension activity is now complete. Since it models the activity at a high level, comparison can be made of the relative costs of any approaches to software comprehension (manual or automatic) that fit the framework.

2.3.1.1 The Processor

There are two primary types of processor: people, and software tools. The characteristics, internal representations, and methods of each type are examined to determine common ideas that can add detail to the framework.

2.3.1.2 People as Processors

There has been a large amount of work undertaken to determine how people understand software, and how their understanding is represented in the mind. The understanding activity is generally termed *program comprehension.*

Novice and expert maintainers understand code differently. Novices adopt a syntactic orientation, organising their knowledge structures around the program syntax, whereas experts organise their knowledge around algorithms and functional characteristics within their domain of expertise [MAYR95]. The comprehension model used initially depends on the maintainer's level of domain knowledge and code familiarity [MAYR94]. Models of program comprehension can be divided into three groups: top-down, bottom-up, and integrated.

Top-down understanding is typically applied when the code is familiar [MAYR95] and a good example of a top-down model is that defined by Soloway, Adelson, and Ehrlich [SOLO84], [SOLO88]. This model views the process of comprehension as the construction of a hierarchy containing goals. These goals are decomposed into structures called plans, which can describe a strategy for achieving a goal, a language independent problem solution, or be a code fragment implementing such a solution. Plans can be decomposed further into lower-level plans [MAYR95]. Brooks presents another top-down approach using a hierarchy of hypotheses [BROO83].

Bottom-up models (typically used when the code is unfamiliar) suggest that the maintainer starts building a mental representation from the source code and chunks together elements into higher order structures. Chunking refers to the process of attaching descriptive labels to knowledge structures at various levels. Chunks can contain lower-level chunks with a description of how they interrelate [MAYR95]. Pennington's model is an example of a bottom-up approach [PENN87].

The integrated approach subsumes the other two types by providing a framework within which both can be used as necessary. This model was suggested by von Mayrhauser and Vans and a number of studies have been performed to validate it [MAYR95] (see also [MAYR97], [MAYR98]).

All of the comprehension models have aspects that appeal to the personal experience of software maintainers. By including elements from top-down and bottom-up methods, the integrated meta-model of von Mayrhauser and Vans appears to be superior to the others. This intuitive assessment is confirmed further by the empirical studies undertaken by von Mayrhauser et al., and by the typical experience of professional maintainers. Glimpses of the meta-model can be seen in the other theories, e.g. to explain the experiences of professional maintainers, Brooks suggests that bottom-up understanding is a degenerate case of top-down understanding [BROO83]. A more plausible explanation for this would seem to lie in the meta-model approach. It is also interesting to note that since the development of the meta-model, no new major comprehension models have been proposed despite the relatively large number produced before its creation.

Von Mayrhauser and Vans identify three major components common to all models of comprehension: a knowledge base, a mental model, and methods for acquiring knowledge [MAYR95]. The knowledge base contains the maintainer's general knowledge of the application domain, software engineering and maintenance knowledge, their experience and skills, and any other knowledge relevant to the task. The mental model is the internal, working representation of the software under consideration [MAYR95]; in other words, it contains the current state of comprehension. The methods for acquiring knowledge (and thus updating the knowledge base and mental model) vary from theory to theory. Littman et al. identify two major strategies: systematic, and as-needed. The systematic approach involves detailed line-by-line study of the program code whereas the as-needed strategy suggests localising the section of program required for a change before understanding it in greater depth [LITT86].

The literature on program comprehension suggests that there are three important elements to be added to the comprehension activity framework: a knowledge base, a mental model, and a collection of methods for acquiring and updating knowledge. Figure 5 shows the comprehension activity framework extended for a person.

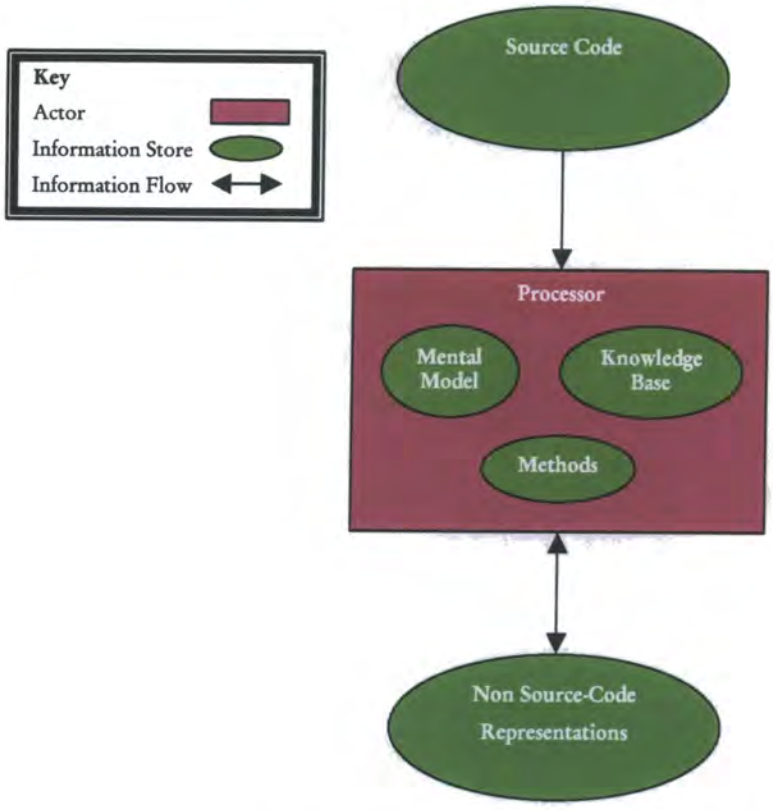


Figure 5: Comprehension Activity Framework for a Person

The framework should describe the comprehension activity regardless of whether the processor is a person or a software tool. To ensure the framework is capable of modelling both types of processor, two tools are now examined. Since this thesis is concerned with concept assignment by plausible reasoning, both are tools that address this problem using such techniques.

2.3.1.3 Software Tools as Processors

This section discusses the common characteristics of two software tools in the context of the comprehension activity framework. Their operational details are discussed in later chapters.

The DM-TAO [BIGG94] and IRENE [KARA92] systems adopt different approaches to locating concepts in source code. IRENE uses an exclusively top-down approach driven from a strong domain model and user input. DM-TAO works in a largely interactive manner and forms part of a larger toolset designed to facilitate design recovery (the DESIRE system). Both have a domain model

(although the degree of formality of knowledge representation varies between them), a representation of their current belief about the software under analysis, and methods for updating their current belief representation and domain model (although the model may be updated by the user rather than automatically). It is clear that these systems have similar characteristics to the psychological phenomena observed by researchers in human program comprehension. The domain model of a software tool corresponds to the knowledge base of a person. Representations of current belief in a tool correspond to a person's mental model and both people and software tools have methods by which they acquire and evaluate new knowledge.

The comprehension activity framework can now be modified to include these ideas. The three components added to the framework for people can be redefined to be valid for both processor types. They are now:

- A *knowledge base* containing the processor's knowledge about the domain, language, and other pertinent information required to perform the comprehension task. This knowledge base would be considerably richer and more flexible for a person than a software tool.
- An *internal representation* to store the processor's current understanding of the source code being analysed. This corresponds to the mental model of a human maintainer.
- *Methods* by which the internal representation and knowledge base can be updated.

These changes are shown in Figure 6.

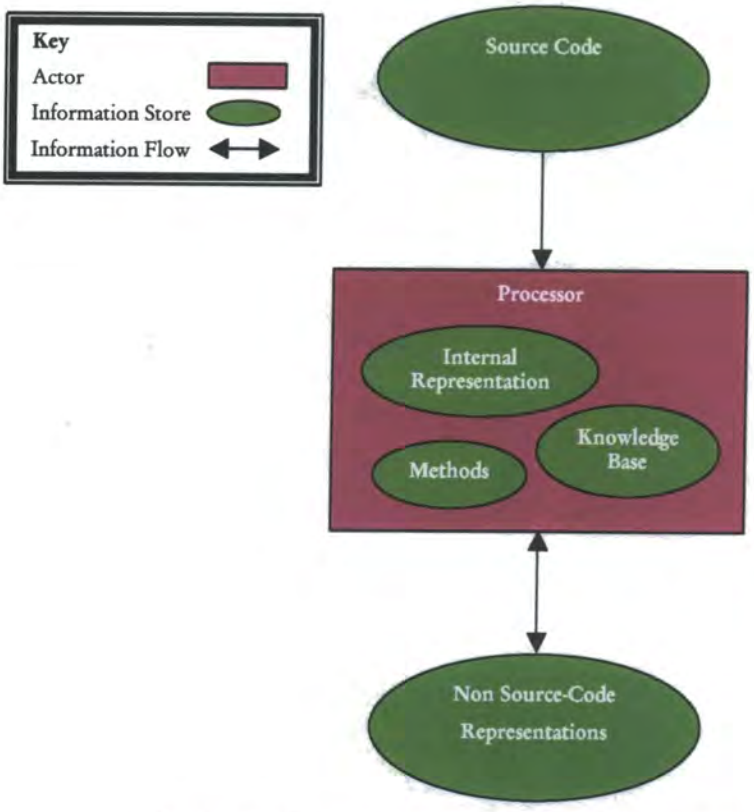


Figure 6: Comprehension Activity Framework with Processor Related Entities

2.3.2 Representations

Since the focus of the work in this thesis is on the concept assignment problem described in Chapter 1, the general framework in Figure 6 can be made more specific with respect to the non source-code representations. The output required of solutions to the concept assignment problem is a collection of domain concept names related to parts of the source code. This is defined as the specific target representation for the framework and is referred to as the *source-label* representation. A domain is defined as a problem area [DEBA94] but Tilley et al. note the over-burdening of the term [TILL96a]. For the purpose of HB-CA, concepts can be drawn from any problem area considered appropriate by the software maintainer. It is likely that both general software-engineering and specific application-domain concepts will be used.

Figure 7 shows an updated framework. Note that the *use* of the source-label representation by the processor is no longer shown on the framework since the

primary concern here is the translation from source code to source-label representation.

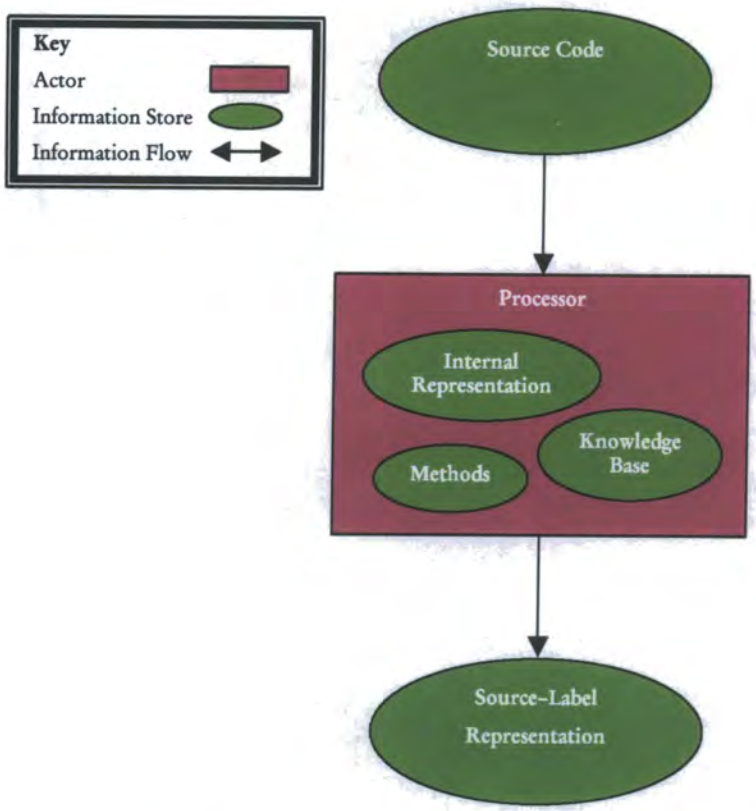


Figure 7: Comprehension Activity Framework with Specific Output Representation for Concept Assignment

These representations can be defined precisely through the definition of their supporting terms and the use of formal notation.

2.3.2.1 Source Representation

The formal model described in this section captures various properties of the source code, allowing reference to its constituent parts as required by the target representation.

The source code can be regarded as a number of lines.

$$\textbf{Source} : \{x : \textbf{Line}\} \qquad (1)$$

Each line is made up of lexemes and is numbered sequentially.

$$\textbf{Line} : (\{y : \textbf{Lexeme}\}, \textit{seqnum} : \textbf{Integer}) \quad (2)$$

Lexemes are the basic units used in parsing and do not need to be specified in more detail. They are described by their start and end character positions (relative to the first character of the source code) and are represented as a string.

$$\textbf{Lexeme} : (\textit{start} : \textbf{Integer}, \textit{end} : \textbf{Integer}, \textit{token} : \textbf{String}) \mid \textit{start} \leq \textit{end} \quad (3)$$

Definitions (1) to (3) establish a formal, lexical representation for the source code.

2.3.2.2 Target Representation

The target representation is a collection of concept names (labels) related to parts of the source code (termed segments). This can be expressed formally.

$$\textbf{TR} : \{(x : \textbf{Segment}, y : \textbf{String})\} \quad (4)$$

A concept is any descriptive term (usually related to computational intent) nominated by the processor to represent some important item or activity within a software-engineering or application domain. This is defined initially as a string.

$$\textbf{Concept} : \textbf{String} \quad (5)$$

A segment is a contiguous group of lines in the source code. A basic definition is shown below.

$$\textbf{Segment} : (\textit{start} : \textbf{Line}, \textit{end} : \textbf{Line})$$

This needs to be extended to capture the notion that *start* must be equal to or less than *end*. A new function φ is defined for "occurs before".

$$\varphi : (\textbf{Line}, \textbf{Line}) \rightarrow \textbf{Boolean}$$
$$\varphi\,((a, b), (c, d)) = b \leq d \qquad (6)$$

The basic definition of segment can now be extended to include this constraint.

$$\textbf{Segment} : (\textit{start} : \textbf{Line}, \textit{end} : \textbf{Line}) \mid \textit{start}\ \varphi\ \textit{end} \qquad (7)$$

Definitions (4) to (7) establish a formal version of the target representation.

The comprehension activity can now be regarded as a function P between the source and target representations.

$$\textbf{P} : \textbf{Source} \rightarrow \textbf{TR} \qquad (8)$$

The processor provides the method by which the mapping under the function takes place.

The comprehension activity framework now contains all the general components required to model the software comprehension activity for concept assignment, whether the processor is a person or software tool. The components are common to both types and later chapters discuss some instances in more detail. Any automatic concept assignment solution set in the context of the framework can be shown to perform the same translation as a person undertaking concept assignment manually. Consequently, the relative costs of the approaches can be compared.

2.4 Summary

This chapter has discussed the software maintenance process and its improvement. A descriptive framework has been presented to model the software comprehension activity and its representations. These representations have been defined formally. The framework and formal model provide the context for the work described in the next few chapters of this thesis.

Chapter 3 presents an outline of the processes and data structures used in the Hypothesis-Based Concept Assignment method, relating these to the comprehension activity framework. It discusses the rationale for the method's design and examines the structure of the knowledge base used by HB-CA, comparing it to those employed in DM-TAO and IRENE. An example source program is presented on which the operation of HB-CA is demonstrated in later chapters.

Chapter 3

Hypothesis-Based Concept Assignment

3.1 Introduction

Chapter 2 presented a framework and formal model to describe various aspects of the software comprehension activity and its associated representations.

This chapter outlines a new approach to solving the concept assignment problem discussed in Chapter 1. It is termed Hypothesis-Based Concept Assignment (HB-CA). The processes and data structures of the method are discussed in the context of the comprehension activity framework described in Chapter 2. A comparison is made between the general characteristics of this method and other plausible reasoning solutions to the concept assignment problem.

A program fragment and knowledge base are presented as an example to illustrate the method's operation in later chapters.

3.2 Characteristics of Concept Assignment Methods

This section discusses general characteristics of two concept assignment methods that address the problem using plausible reasoning. Table 1 shows a summary of areas on which these are compared, with lengthy discussion reserved for later chapters. Hatched boxes show where a method's characteristic is shared with HB-CA.

	HB-CA	DM-TAO (Conceptual *grep*)	DM-TAO (Conceptual Highlights)	DM-TAO (Identification)	IRENE
Direction	Bottom-Up	Top-Down	Bottom-Up	Restricted Bottom-Up	Top-Down
Interactivity	None	High	None	High	High
Knowledge Base Representation	Semantic Network	Semantic/Connectionist Hybrid Network			Formal Relations
Knowledge Base Complexity (Concept Types)	Low	High			Low
Knowledge Base Complexity (Relationship Types)	Low	High			Low
Knowledge Base Creation Cost	Low	High			Medium
Knowledge Base Update Method	Manual	Semi-Automatic			Manual
Knowledge Base Update Cost	Low	High			Medium
Initial Information Source	Source Code	User	Source Code	Source Code	User
Initial Information	Textual Indicators	User-Supplied Concept, Syntactic, Lexical, and Clustering Clues	Syntactic, Lexical, and Clustering Clues	Syntactic, Lexical, and Clustering Clues	User-Supplied Hypothesis
Clustering Method	Self-Organising Map	Feature Extractors			Unspecified
Clustering Data Used	Hypotheses	Syntactic Features			Unspecified
Concept Binding Evidence	Hypotheses	Syntactic Features			Syntactic Features/ Domain Model
Concept Binding Method	Scored Weight of Evidence with Disambiguation Rules	Connectionist Network Triggering and Propagation			Plausibility Measure using Weighted Matching Rules
Explanatory Power	Medium	Low			High

Table 1: Characteristics of Concept Assignment Methods

3.3 The Hypothesis-Based Concept Assignment Method

The Hypothesis-Based Concept Assignment method is a three-part non-interactive process. It operates on the procedure division of IBM COBOL II programs (although a complete program is provided as input). Chapter 1 discusses the reasons for using only the procedure division.

The three stages of HB-CA are:

- Hypothesis Generation
- Segmentation
- Concept Binding

In terms of the comprehension activity framework, these parts reside in the methods oval as shown in Figure 8.

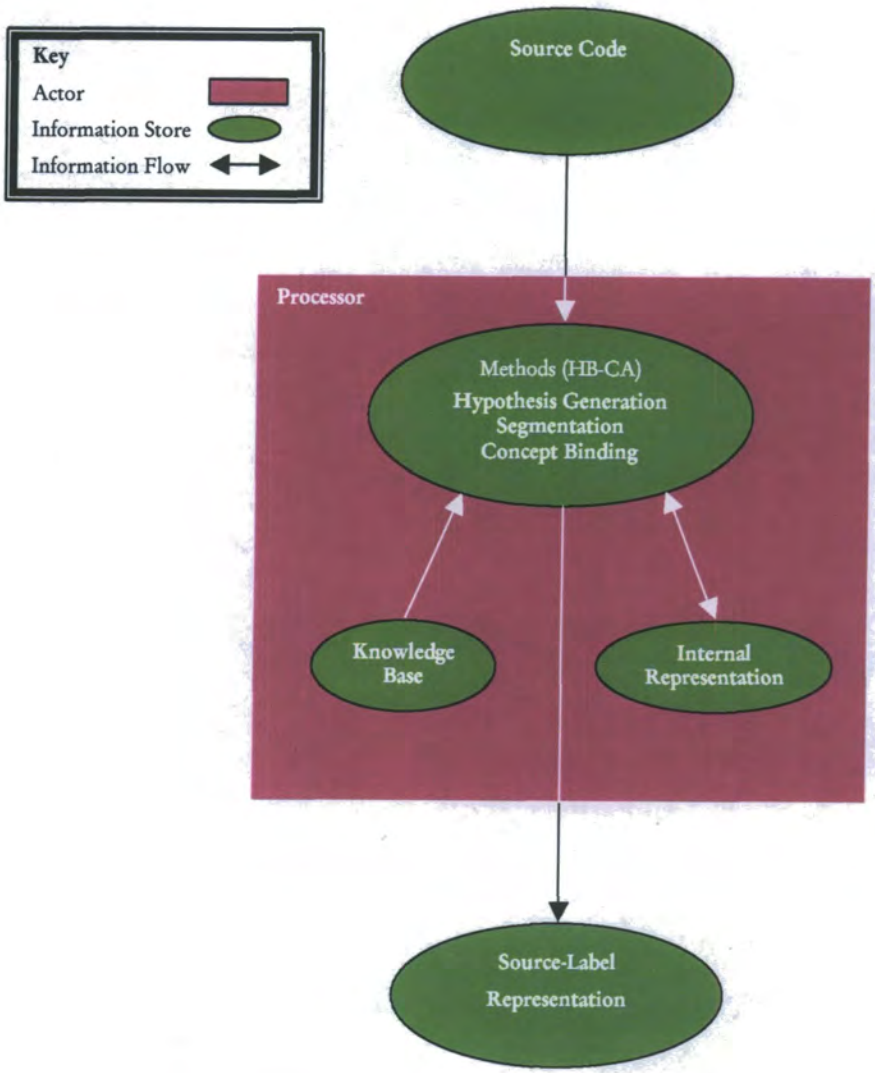


Figure 8: Comprehension Activity Framework Showing HB-CA Processes

The flow of control and data is sequential. The process begins with hypothesis generation from source code. This is followed by segmentation of the hypotheses to determine regions of conceptual focus in the program. Finally, concept binding finds the dominant concept in each segment. Figure 9 shows the sequence of processing with the internal representations used by each stage.

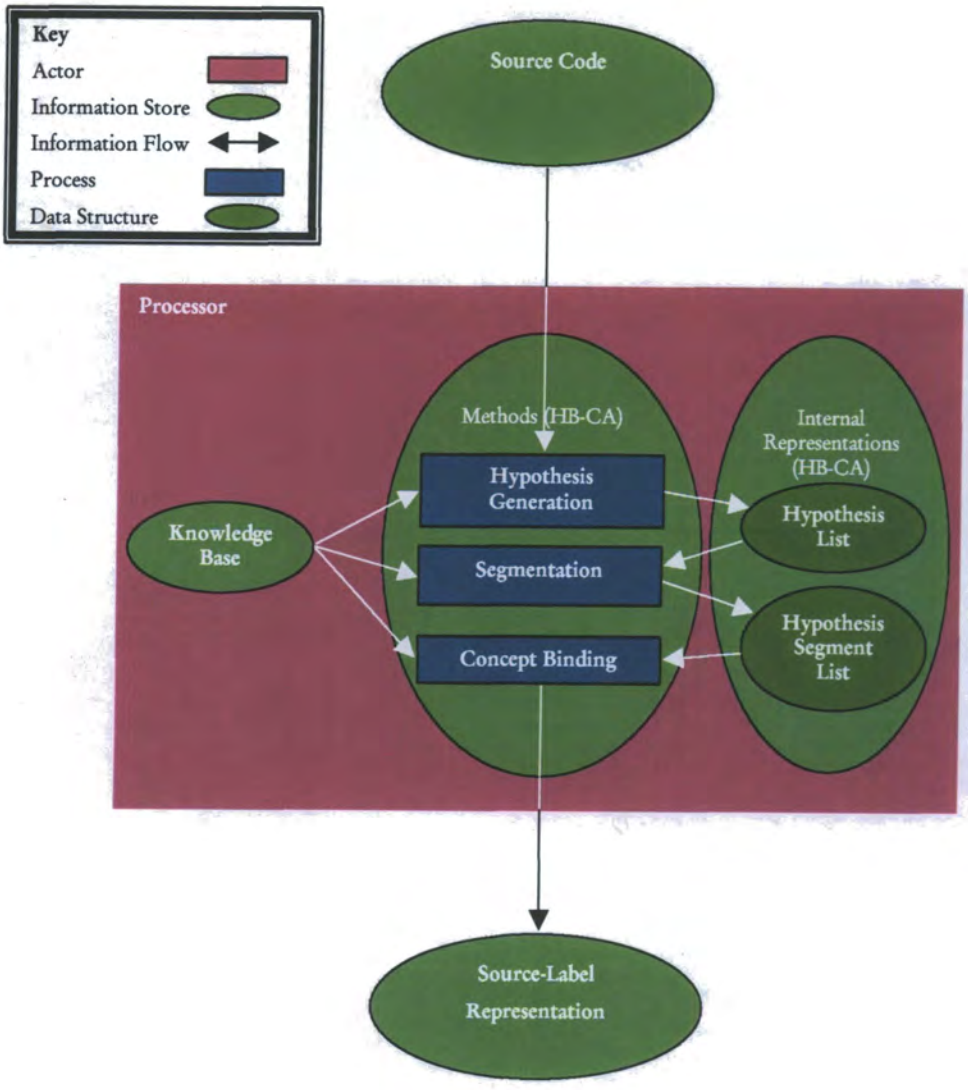


Figure 9: Comprehension Activity Framework Showing HB-CA Processes and Internal Representations

3.3.1 Hypothesis Generation

The hypothesis generation stage takes source code as its input. Using information contained in the knowledge base, it scans the source code for indicators of various concepts. When an instance is found and matched, a hypothesis for the appropriate concept is generated. Matching is performed using a variety of flexible criteria. The resulting collection of hypotheses is ordered by the position of the indicators in the source code.

3.3.2 Segmentation

The segmentation stage takes the sorted hypotheses and attempts to break them into segments. Initially, this is performed using hypotheses for primary segmentation points (COBOL II section boundaries). Each of the initial segments is analysed to determine whether it has the potential to contain a number of smaller segments. If this is the case, a self-organising map is used to establish areas of conceptual focus within the segment. These areas are analysed and smaller segments created if necessary. The output of the stage is a collection of segments, each containing a number of hypotheses.

3.3.3 Concept Binding

This stage analyses each segment's hypotheses to determine which concept has the most evidence. It exploits relationships in the knowledge base to generate conclusions, and scores these on the basis of concept occurrence. A number of disambiguation rules can be applied to choose between equally strong concepts. When a concept has been selected, the segment is labelled with the name of that concept. After all segments have been analysed and labelled, the results form the overall output of the method.

3.4 Characteristics of Concept Assignment Methods

Each of HB-CA's stages is described in detail in the next few chapters but it is useful to compare some of its general characteristics with those of DM-TAO and IRENE, which have both addressed the concept-assignment problem using plausible reasoning.

The methods are compared using two characteristics: direction of operation, and interactivity.

3.4.1 Direction of Operation

The direction of operation of a method describes whether it works top-down (from the knowledge base to the code), bottom-up (from the code to the knowledge base), or a combination of both.

HB-CA adopts a purely bottom-up approach, developing a hypothesis representation from the source code. This is used for segmentation and concept binding in the latter stages of the method.

IRENE is based on a top-down approach (see [KARA92]). It is driven by user-supplied hypotheses that are analysed by the system. Derivations and dependencies in the knowledge base are used to infer the existence of other concepts, and IRENE attempts to find implementations of these in the source code.

DM-TAO can operate in several ways: top-down where a user specifies a concept for search (conceptual *grep*), bottom-up where all instances of any concept in the knowledge base are found (conceptual highlights), and "directed" bottom-up where a concept assignment is proposed for user-selected source code (identification) [BIGG94].

The different approaches to the direction of operation are summarised in Table 2.

	HB-CA	DM-TAO (Conceptual *grep*)	DM-TAO (Conceptual Highlights)	DM-TAO (Identification)	IRENE
Direction	Bottom-Up	Top-Down	Bottom-Up	Restricted Bottom-Up	Top-Down

Table 2: Characteristics of Concept Assignment Methods - Direction of Operation

The advantage of a purely bottom-up approach is that a simpler knowledge base can be used than that required for top-down approaches. A bottom-up approach

performs most of its understanding based on information from the source code. If the source code is hard for a person to understand (due to a lack of meaningful items within it), it will probably be hard for a bottom-up concept assignment system. Top-down systems may be able to avoid this problem by performing most of their inference using the domain model. This requires more investment in the creation and maintenance of the knowledge base, as it is the primary understanding mechanism.

DM-TAO's bottom-up mode has the same goal as HB-CA: to provide a list of all recognised concepts to the user. The methods of understanding and presentation differ. The DM-TAO method also forms part of the DESIRE toolset intended for supervised use (see [BIGG89], [BIGG93], [BIGG94]) whereas HB-CA is intended to operate unassisted.

3.4.2 Interactivity

Interactivity is the amount of user-involvement required in the concept assignment process.

HB-CA is a non-interactive method requiring no user involvement, other than prior creation of the knowledge base.

IRENE is highly interactive. The concept search process is initiated from user-supplied hypotheses with the system making further suggestions. These are verified by the user for IRENE to continue its analysis.

DM-TAO can operate with various levels of interactivity. It is user-driven in both top-down mode (conceptual *grep*), and in "directed" bottom-up mode (identification). In bottom-up mode (conceptual highlights), it is non-interactive, although the expectation is that the results will be employed by the user to extend the search further.

Table 3 summarises the level of interactivity required by the different approaches.

	HB-CA	DM-TAO (Conceptual *grep*)	DM-TAO (Conceptual Highlights)	DM-TAO (Identification)	IRENE
Interactivity	None	High	None	High	High

Table 3: Characteristics of Concept Assignment Methods - Interactivity

The level of interactivity can have an effect on the overall cost of the method. Since interactive methods require a user to supervise their operation and guide them in the understanding task, they can incur a high cost. Non-interactive methods can perform their analysis unassisted and therefore are cheaper to execute. The reduced cost of using an unsupervised method is balanced by the potentially more accurate analysis of an assisted approach. The latter may allow the maintainer to get relevant information more quickly as the search space for the tool is reduced. Unassisted approaches may produce more irrelevant information but the maintainer does not waste time waiting for analysis to be performed.

Unassisted approaches require the solution of at least one additional problem: finding the location and extent of a concept implementation. HB-CA's solution to this is discussed in Chapter 5. Assisted approaches do not have to deal with this situation to the same degree, as the user can suggest or verify the position and size of concept implementations.

3.5 Knowledge Base

In order to discuss the constituent processes of HB-CA in detail, the structure of the knowledge base needs to be established. The knowledge base used for HB-CA shares a number of similarities with those used in IRENE and DM-TAO. All have the notion of a concept: a descriptive term to be attached to some part of the source code being analysed. They all store evidence for a concept expressed in the implementation language, and they all have ways of relating concepts to other concepts.

The knowledge base used in HB-CA is termed the library.

It is anticipated that the user, or some other person responsible for knowledge base maintenance, will construct the library, possibly using automated assistance such as that described in [SAYY97]. This would take place before the first use of HB-CA and the knowledge base content then could be improved as the user gained experience. Section 9.2.2 shows a model that includes this feedback process.

3.5.1 Knowledge Representation in the Library

Knowledge in the library can be represented as a semantic network. Semantic networks are graph structures consisting of nodes, and labelled arcs that describe the relationships between the nodes [KUWA97]. The library nodes also have attributes that are explained below.

There are two entities in the library that are represented as nodes in a semantic network: concepts, and indicators. Concepts are the terms nominated by the user to describe items or activities in the domain. Indicators are evidence for concepts expressed in the implementation language, in this case IBM COBOL II.

The library encodes two types of relationship:

- Indicator-Concept
- Concept-Concept

The indicator-concept relationship maps evidence for a concept to that concept. Concept-concept relationships map concepts to others to form composites and specialisations.

3.5.1.1 Indicators

Indicators have a number of attributes:

- Name
- Class
- Data

The *name* is a string used within the library to identify the indicator and provide an abstraction from the actual data. The *class* refers to the type of feature represented (e.g. identifier). This allows the indicator recognition process to filter indicators in the library for those appropriate to the search method being employed. The *data* is the actual evidence to be found in the source code. Alternatively, it may be a reference to another container for the data. The latter would be appropriate for complex indicators such as code fragments.

Indicators are represented in the semantic network diagrams throughout this thesis as shown in Figure 10.

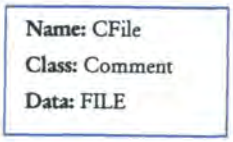


Figure 10: Example of an Indicator in Semantic Network Representation

Formal definitions of indicator and class, in the context of the model presented in Chapter 2, are given below.

$$\textbf{Class : String}$$
$$\forall X : \textbf{Class}, X \in \{\text{"Identifier"}, \text{"Keyword"}, \text{"Comment"}, \text{"Segment Boundary"}\} \quad (9)$$

$$\textbf{Indicator : } (n : \textbf{String}, c : \textbf{Class}, d : \textbf{String}) \quad (10)$$

3.5.1.2 Concepts

Concepts have three attributes:

- Name
- Type
- Level

The *name* is a string to identify the concept, i.e. the nominated descriptive term.

The *type* is either *action* or *object*. Action concepts are those that do something (typically, the name of an action concept is a verb, e.g. Read). Object concepts are those things on which action concepts operate (typically, the name is a noun, e.g. File). Classifying concepts by type has improved the operation of the HB-CA method by preventing over-production of hypotheses. Chapter 4 discusses this issue in more detail. The classification also allows greater control of the concept binding search (see Chapters 5 and 6). Additionally, in combination with the relationships described below, it can help to reduce the size of the knowledge base required to represent complex concepts. Concept typing is used by various methods including DM-TAO (see [BIGG93]).

The *level* is either *primary* or *secondary*. Primary concepts represent the most general form of a particular concept; secondary concepts represent more specialised forms of primary concepts, e.g. File might be primary, MasterFile might be secondary. This information is required to help the method degrade its performance gracefully in the event of conflicting evidence. It allows the search methods to select a more general form of a concept if the evidence for specific versions is ambiguous.

Semantic network diagrams in this thesis represent concepts as shown in Figure 11.

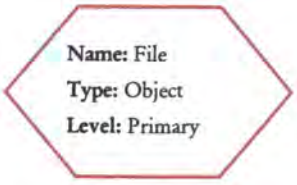


Figure 11: Example of a Concept in Semantic Network Representation

Formal representations of type, level, and concept are shown below.

$$\textbf{Level : String}$$
$$\forall X : \textbf{Level},\ X \in \{\text{"Primary"}, \text{"Secondary"}\} \tag{11}$$

$$\textbf{Type : String}$$
$$\forall Y : \textbf{Type},\ Y \in \{\text{"Action"}, \text{"Object"}\} \tag{12}$$

$$\textbf{Concept} : (n : \textbf{String},\ l : \textbf{Level},\ t : \textbf{Type}) \tag{13}$$

Note that definition 13 extends the original definition of concept (definition 5) to include the additional attributes required by the knowledge base.

3.5.1.3 Indicator-Concept Relationship

The indicator-concept relationship, termed *indicates*, is formed by joining indicators to the concepts for which they provide evidence. An example semantic network showing the indicates relationship is presented in Figure 12.

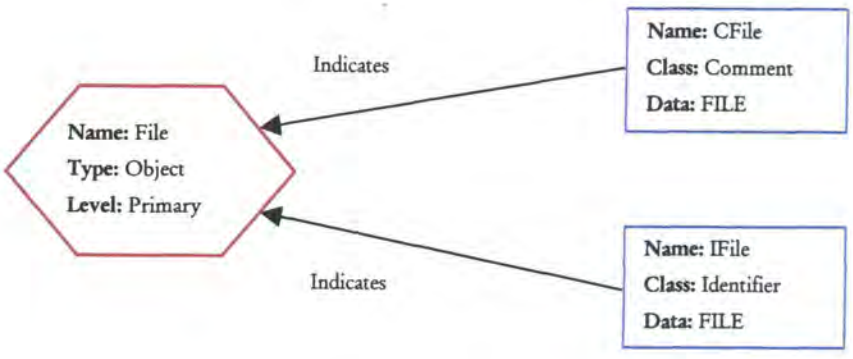


Figure 12: Example of a Semantic Network Showing the Indicates Relationship

Indicates can be stated formally thus:

$$\textbf{Indicates} : \{(p : \textbf{Indicator},\ q : \textbf{Concept})\} \tag{14}$$

3.5.1.4 Concept-Concept Relationships

There are two concept-concept relationships in the library: *composition*, and *specialisation*. The formal model can capture the general form of concept-concept relationships thus:

$$\mathbf{CCR} : \{r \mid r : \{(a : \mathbf{Concept}, b : \mathbf{Concept})\}\} \tag{15}$$

Composition and specialisation form instances of the CCR relation.

Specialisation relationships are formed by linking secondary concepts (i.e. specialisations) to primary or other secondary concepts. This is indicated on a semantic network diagram using a dashed arrow (see Figure 13). If X is a specialisation of Y then the arrow will point from X to Y. Although the library structure permits secondary actions to be defined, the HB-CA method presented here does not support their use.

Figure 13: Example of a Semantic Network Showing the Specialisation Relationship

Note the use of a dashed arrow to indicate specialisation.

It is extremely important that general forms of object concepts do not share specialised concepts. The path from a specialised concept back to its more general forms should always be unambiguous. This is because the concept assignment methods have no way of handling more than one general version of a particular specialised concept, although they can deal with multiple layers of specialisation. The suggested structure is a tree with the most general form of a concept at its root and specialisations extending from it. There should only be one path from a leaf node to the root, but this does not have to be a single step. Thus, concepts can be part of a chain of specialisation, each concept having only one general form but

potentially many specialisations. If a specialised concept is required by more than one general form, an additional concept should be added to the library to represent the entities separately. Examples of acceptable and unacceptable structures for the specialisation relationship are shown in Figure 14. The red lines denote the links that would cause problems.

Unacceptable Acceptable

Figure 14: Examples of Acceptable and Unacceptable Forms of the Specialisation Relationship

The specialisation relationship can be stated formally:

$$\textbf{Specialisation} : \{((a,b,c):\textbf{Concept}, (d,e,f):\textbf{Concept}) \mid e = \text{"Secondary"}\} \quad (16)$$

Composition relationships are formed by creating composite nodes in the semantic network to join primary action concepts to primary object concepts. This forms an action:object structure (essentially a verb and noun construction) to convey more information to the user (e.g. Read:File rather than merely Read).

Creating a composition of two primary concepts also produces a series of implied composites with all specialisations of the primary object concept. These are not stored in the semantic network but are used as required by the segmentation and concept binding methods. Figure 15 shows an example semantic network with a composite concept.

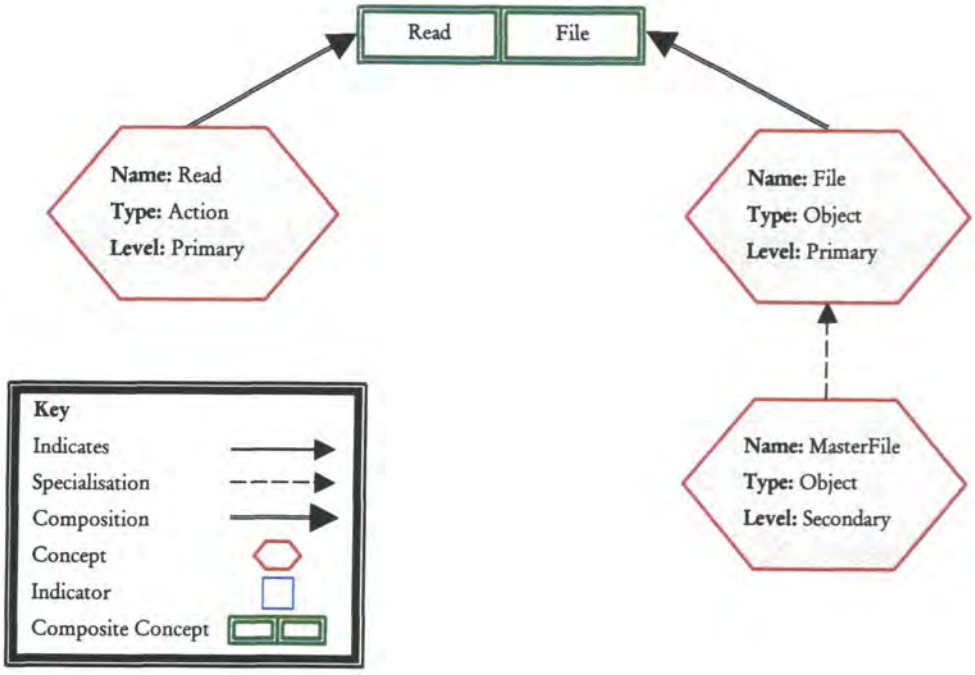


Figure 15: Example of a Semantic Network Showing the Composition Relationship

In Figure 15, a composite concept, Read:File, is formed from the Read and File concepts. The implication mechanism discussed above means that an implied composite, Read:MasterFile, also would be formed although not stored explicitly.

Composite concepts do not have their own indicators (the indicators of the two constituents form the evidence for the composite).

Composition can be expressed formally:

$$\textbf{Composition:}\{((a,b,c)\textbf{:Concept}, (d,e,f)\textbf{:Concept}) \mid b = \text{"}Primary\text{"}, \; e = \text{"}Primary\text{"}, \; c = \text{"}Action\text{"}, \; f = \text{"}Object\text{"}\} \tag{17}$$

The formal representations defined in expressions (9) to (15) can be combined to give a formal definition of the knowledge base.

$$\mathbf{KB} : (\{x : \mathbf{Concept}\}, \{i : \mathbf{Indicator}\}, \{(p : \mathbf{Indicator}, q : \mathbf{Concept})\}, \{r \mid r : \{(a : \mathbf{Concept}, b : \mathbf{Concept})\}\}) \quad (18)$$

The HB-CA library structure can be expressed as an instance of the general KB type.

$\mathbf{L} : \mathbf{KB}$

$\mathbf{L} = (C, I, Z, R)$

$I = dom\ Z$

For some T: **Specialisation**, $T \in R$

For some P: **Composition**, $P \in R$

3.6 Knowledge Base Characteristics

This section compares various characteristics of the knowledge bases used in IRENE, DM-TAO, and HB-CA. The characteristics examined are the costs of creation and maintenance, and the knowledge base complexity, as measured by the number of inter-concept relationships and their types. A brief description of the IRENE and DM-TAO knowledge bases is presented in sections 3.6.1 and 3.6.2.

3.6.1 DM-TAO Knowledge Base

The knowledge base and inference engine of DM-TAO are combined into one structure. It uses a connectionist-based inference engine [BIGG93]. The knowledge base is a domain model built as a network, in which each concept is represented as a node and inter-concept relationships are modelled as explicit links between the nodes. HB-CA's library is similar to this. Each concept has associated information regarding the features that characterise it, its relationships to other domain concepts, and informal knowledge such as programmer terminology. The syntactic and conceptual context in which the concept occurs also may be stored. The domain model captures the underlying semantics in the target domain through a rich set of inter-concept relationships, embodying the nature and degree of semantic association between domain concepts [BIGG93]. The network is

organised in layers of abstraction and contains many types of node. These are connected by several types of inter-node link, which have real-valued weights associated with them to quantify the strength of the relationship. The weights are updated semi-automatically in response to user evaluation of the correctness of concept assignments.

3.6.2 IRENE Knowledge Base

IRENE's knowledge base models the domain by using concepts and two major relations between them: derivation, and dependency [KARA92]. Derivation captures the notion that a concept X is derived from a concept Y if there is a function f such that f(Y) = X. Dependency is similar but the function f is unknown. These relations are transitive. There are two other relations used in the IRENE knowledge base: strong and weak implication. Strong implication captures the expectation of the existence of a concept when knowing the existence of another concept. Weak implication expresses the plausibility that a concept exists, upon the knowledge that its implying concept exists. The implication relations are intransitive. The domain model also stores possible concept realisations in a COBOL program, dividing IRENE's knowledge into software-dependent and software-independent categories.

3.6.3 Knowledge Base Complexity

The complexity of the knowledge bases can be compared using the number of concept types and the number of inter-concept relationships employed to represent knowledge.

HB-CA employs two inter-concept relationships and two types of concept. This provides a relatively simple knowledge base capable of representing a wide range of concepts. The HB-CA library is effectively a reflection of the maintainer's current domain understanding and interest.

DM-TAO has the most complex knowledge base employing a large number of concept and relationship types. This allows the system to perform powerful inference but at the expense of greater maintenance than the HB-CA library. The way that DM-TAO updates its knowledge means that the knowledge base does not

reflect the maintainer's understanding in the same way as HB-CA, but forms its own "understanding" of the domain.

IRENE's knowledge base is of similar complexity to HB-CA, utilising four inter-concept relations. The restrictions on the application of these are considerably greater than HB-CA since they require a formal relationship to hold between the concepts. IRENE does not differentiate between types of concept.

There are advantages to each approach. Simpler approaches, such as HB-CA and IRENE, allow the knowledge base to be created and maintained easily by a user. The more complex approach of DM-TAO makes this a difficult activity but provides a subtler inference system. Its knowledge base is updated automatically although a user is still required to assess the validity of concept assignments. The difficulty of creating and maintaining such a knowledge base may have contributed to the fact that DM-TAO has not moved beyond a research prototype (see [BIGG93], [BIGG94]). The formal relations employed by IRENE may incur a higher initial cost than HB-CA when the domain model is created. HB-CA is capable of concept assignment using minimal information.

The approach taken to source-code evidence can also have an impact on cost. DM-TAO and HB-CA use feature analysers with flexibility in the recognition methods. This reduces the need to store a large range of specific implementation evidence. The examples of concepts shown in [KARA92] imply that IRENE requires a larger range of code examples to match source code features. The differences between the systems are summarised in Table 4 below, and Table 1 in section 3.2.

	HB-CA	DM-TAO (Conceptual *grep*)	DM-TAO (Conceptual Highlights)	DM-TAO (Identification)	IRENE
Knowledge Base Representation	Semantic Network	Semantic/Connectionist Hybrid Network			Formal Relations
Knowledge Base Complexity (Concept Types)	Low	High			Low
Knowledge Base Complexity (Relationship Types)	Low	High			Low
Knowledge Base Creation Cost	Low	High			Medium
Knowledge Base Update Method	Manual	Semi-Automatic			Manual
Knowledge Base Update Cost	Low	High			Medium

Table 4: Characteristics of Concept Assignment Methods - Knowledge Base

3.7 Example

This section shows a fragment of real-world COBOL II (Figure 16), and an example knowledge base expressed as a semantic network (Figure 17). These are used in the next three chapters to illustrate the operation and data structures of the constituent parts of HB-CA.

3.7.1 COBOL II Fragment

```
GB21    C00-READ-APS SECTION.                                          0193
GB21    C00-000.                                                       0194
GB21   *    READ APS MASTER FILE                                       0195
GB21        CALL 'GBAAY0X' USING APS-RECORD-IN.                        0196
GB21        IF APS-EOF = END-OF-FILE                                   0197
GB21                MOVE HIGH-VALUES TO APS-RECORD-IN                  0198
GB21                GO TO C00-999.                                     0199
GB21        MOVE '1' TO W-GBCM0133-2.                                  0200
GB21        CALL 'GBCM0133' USING APS-RECORD-IN W-GBCM0133-2.          0201
GB21    C00-999.                                                       0202
GB21        EXIT.                                                      0203
GB21        SKIP3                                                      0204
GB21    C10-WRITE-APS SECTION.                                         0205
GB21   *    WRITE APS MASTER FILE                                      0206
GB21        MOVE '2' TO W-GBCM0133-2.                                  0207
GB21        CALL 'GBCM0133' USING APS-RECORD-OUT W-GBCM0133-2.         0208
GB21        CALL 'GBAAZ0X' USING APS-RECORD-OUT.                       0209
GB21    C10-999.                                                       0210
GB21        EXIT.                                                      0211
GB21        SKIP3                                                      0212
GB21    C20-PRINT SECTION.                                             0213
GB21    C20-000.                                                       0214
GB21   *    PRINT PECULIAR RECORDS TO BE MANUALLY CHECKED              0215
GB21        IF A-LINENO LESS THAN 25                                   0216
GB21                GO TO C20-010.                                     0217
GB21                                                                   0218
GB21        ADD 1 TO A-PAGENO.                                         0219
GB21        MOVE A-PAGENO TO H1-PAGE.                                  0220
GB21        MOVE C-1 TO P-CC.                                          0221
GB21        MOVE H1-HEADLINE TO P-LL.                                  0222
GB21        PERFORM S00-PRINT.                                         0223
GB21                                                                   0224
GB21        MOVE WS-2 TO P-CC.                                         0225
GB21        MOVE H1-HEADLINE TO P-LL.                                  0226
GB21        PERFORM S00-PRINT.                                         0227
GB21        MOVE 0 TO A-LINENO.                                        0228
GB21                                                                   0229
GB21    C20-010.                                                       0230
GB21        MOVE WS-2 TO P-CC.                                         0231
GB21        MOVE GBAIA010 TO P1-KEY.                                   0232
GB21        MOVE P1-DATALINE TO P-LL.                                  0233
GB21                                                                   0234
GB21        PERFORM S00-PRINT.                                         0235
GB21        MOVE SPACES TO P-LL.                                       0236
GB21        ADD 2 TO A-LINENO.                                         0237
GB21    C20-999.                                                       0238
GB21        EXIT.                                                      0239
GB21        EJECT                                                      0240
GB21    S00-PRINT SECTION.                                             0241
GB21    S00-000.                                                       0242
GB21   *    PRINTS A LINE                                              0243
GB21                                                                   0244
GB21        CALL 'PRINT' USING P-PRINTLINE.                            0245
GB21    S00-999.                                                       0246
GB21        EXIT.                                                      0247
```

Figure 16: Example COBOL II Program Fragment

3.7.2 Example Library Content (Semantic Network)

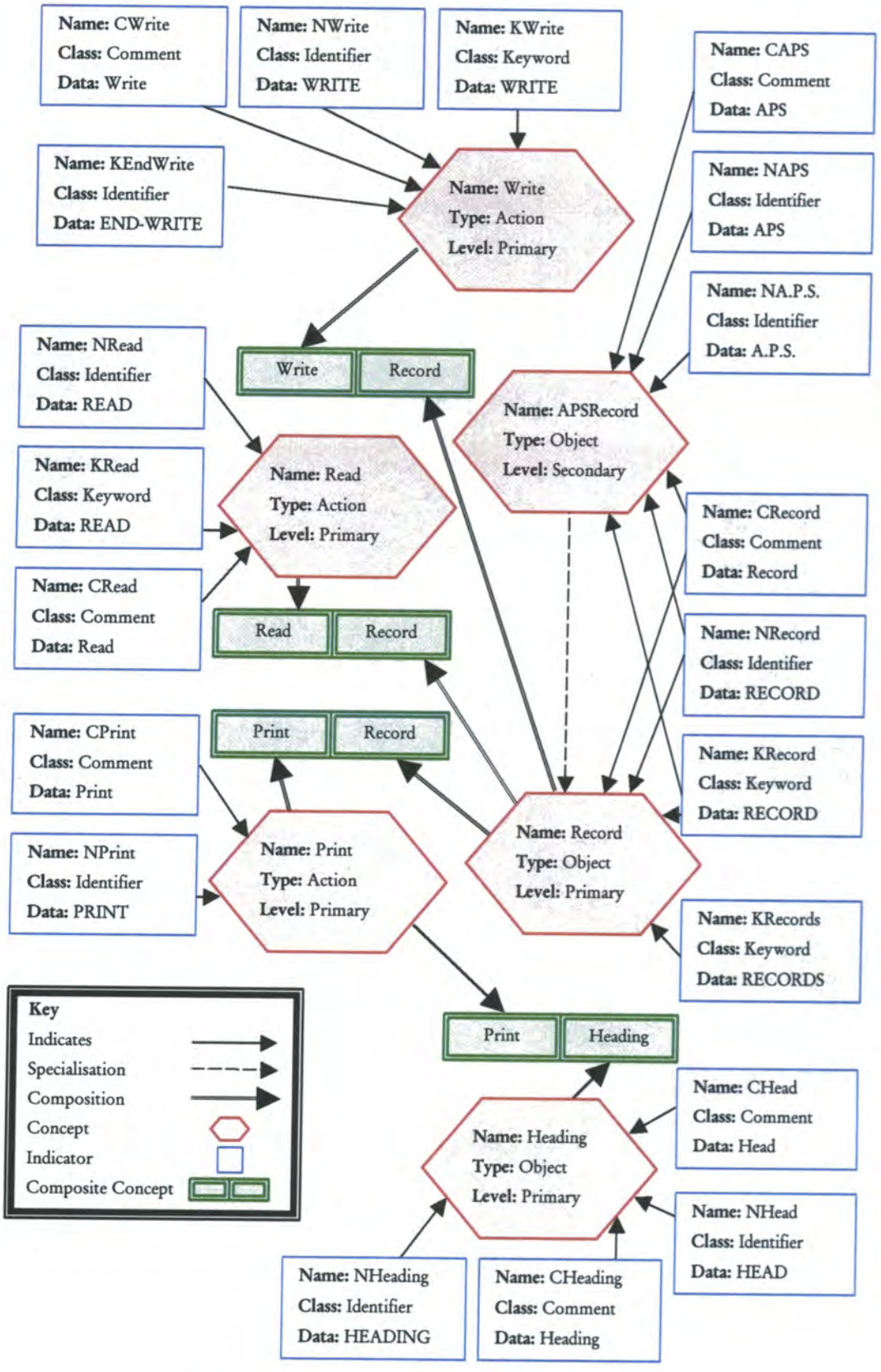


Figure 17: Example Library Content

3.8 Summary

This chapter has described a new approach to concept assignment termed Hypothesis-Based Concept Assignment. A general description of its data structures and processes has been presented, with reference to the comprehension activity framework described in Chapter 2. The formal model has been extended to capture certain characteristics of the HB-CA method and its representations. An example program and knowledge base have been presented for use later in the thesis.

Chapter 4 describes the first stage of HB-CA in detail, further extending the formal model of the approach. Where appropriate, comparisons are made with the IRENE and DM-TAO systems.

Chapter 4
Hypothesis Generation

4.1 Introduction

Chapter 3 provided a general introduction to Hypothesis-Based Concept Assignment and made comparisons with other approaches taken to the concept assignment problem. The overall process of HB-CA was described and the knowledge base was discussed in detail.

This chapter describes the first stage of HB-CA: hypothesis generation. The formal model developed in preceding chapters is extended and a representation for hypotheses defined. Hypothesis generation accepts source code as input and transforms it to a hypothesis list for output.

4.2 Hypothesis Generation

The purpose of this stage is to create an initial conceptual interpretation of the program being analysed.

Hypothesis generation uses the indicator-concept relationship in the knowledge base. When a recognisable indicator is found, a hypothesis is created for each concept that is related to the indicator. These are stored for later use.

The formal model presented in Chapters 2 and 3 can be extended to capture the notion of a hypothesis:

$$\textbf{Hypothesis} : (i : \textbf{Indicator}, c : \textbf{Concept}, l : \textbf{Lexeme}) \tag{19}$$

This representation stores the indicator and concept (essentially a single element of the indicator-concept relationship), and the lexeme that produced the hypothesis. This means that the hypothesis can be linked to its source code origins when necessary.

The white oval in Figure 18 shows where the internal representation produced by hypothesis generation fits in the comprehension activity framework.

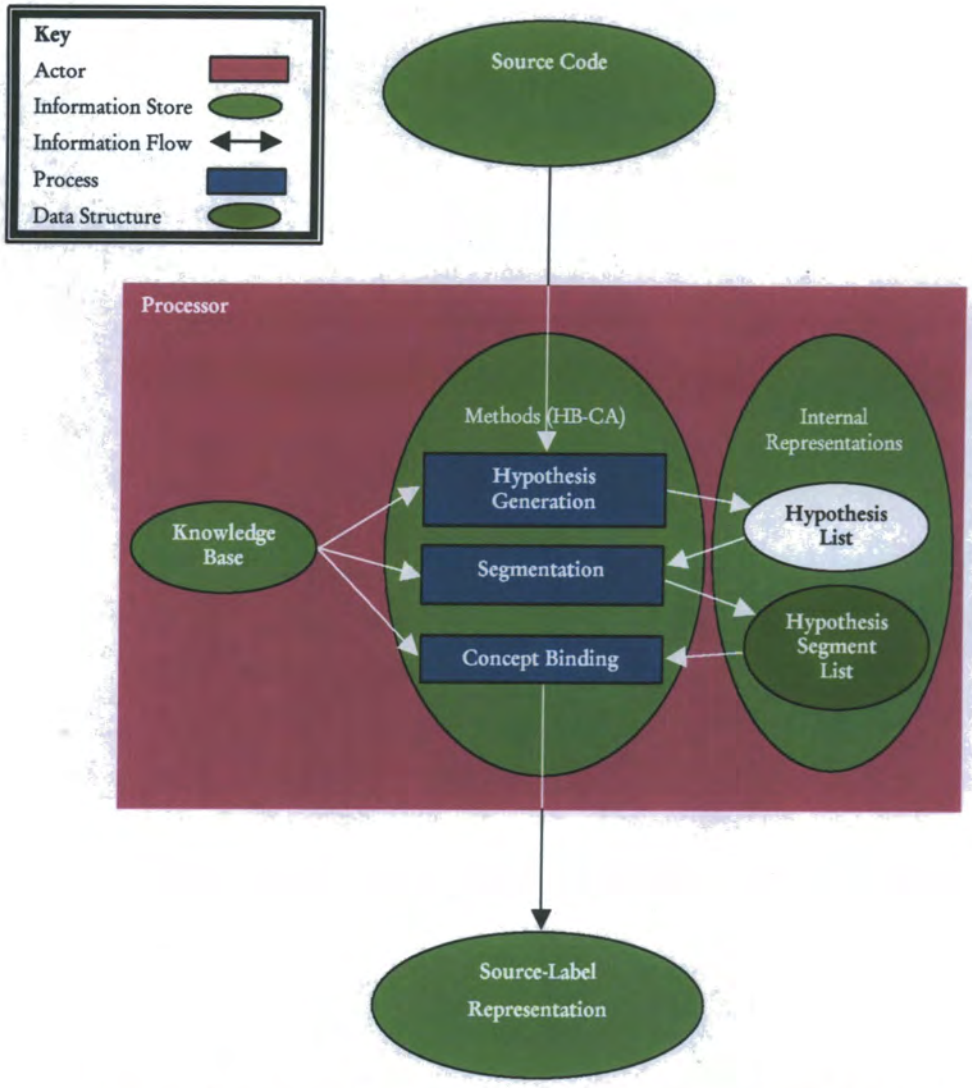


Figure 18: Comprehension Activity Framework Showing the Internal Hypothesis Representation

This representation is termed the *hypothesis list* and is expressed formally as:

$$\textbf{Hypothesis List} : \{h : \textbf{Hypothesis}\} \tag{20}$$

The process of hypothesis generation can be regarded as a function, mapping the source code to a hypothesis list.

$$\mathbf{HG : Source \rightarrow Hypothesis\ List} \qquad (21)$$

Strictly, the knowledge base should be passed as an additional argument to the function. However, since its internal representations are not used elsewhere within the formal model, it is omitted from these definitions.

4.3 Indicator Recognition

This is the key activity of the hypothesis generation stage. Indicators were introduced in section 3.5.1.1 and can take many forms. Table 5 shows Brooks' suggestions for indicators, used in his theory of top-down program comprehension [BROO83].

Internal to the program text	
1	Prologue comments, including data and variable dictionaries
2	Variable, structure, procedure and label names
3	Declarations or data divisions
4	Interline comments
5	Indentation or pretty-printing
6	Subroutine or module structure
7	I/O formats, head, and device or channel assignments
External	
1	Users' manuals
2	Program logic manuals
3	Flowcharts
4	Cross-reference listing
5	Published descriptions of algorithms or techniques

Table 5: Indicators for the Meaning of a Program [BROO83]

Brooks claims that the particular indicators used will vary from maintainer to maintainer, and their relative importance will be different depending on the context

of their use [BROO83]. In HB-CA, these variations are ignored and all indicators are treated with equal weight.

HB-CA only works with internal indicators drawn from types 2, 4, and 6 in Table 5. The method is designed to accommodate additional indicator types without changes to the segmentation or concept binding methods. This is facilitated by merging the output of each type of indicator recognition. Brooks suggests that stereotypical code fragments may be used as indicators [BROO83]. This is a good example of a complex indicator type that would require advanced recognition routines and representations. The indicators used by HB-CA are simple text strings.

Various authors have investigated the contribution of certain types of indicators to the understanding process, and how software maintainers use them. Much work has been performed suggesting the use of code fragments as indicators (also termed beacons by many authors), e.g. [WIED91] and [WIED86]. Gellenbeck and Cook found that meaningful procedure and variable names, typographic signalling, header comments, and mnemonic module names assisted comprehension [GELL91a], [GELL91b]. These findings are confirmed by Teasley's work on naming style, although meaningful names were found to help experts less than novices. Experts used other information sources in the absence of good naming [TEAS94]. Miara et al. investigated the effect of indentation and discovered that a moderate level (2-4 spaces) could help with program comprehension [MIAR83]. The indicators used for HB-CA were chosen partly on the basis of these investigations, and partly for practical reasons, as textual indicators are amenable to simple extraction and analysis by parsers.

In summary, there is evidence to support the use of a variety of indicators when analysing a program for concept assignment. These include code fragments, variable names, module names, procedure names, comments, indentation, and structural information. Analysing a program for simple types of indicator can be performed easily, e.g. using a parser to extract variable names. Complex indicators such as code fragments may require the use of advanced recognition methods.

4.3.1 Indicator Types in HB-CA

There are four indicator types defined in HB-CA:

- Identifiers
- Keywords (programming language reserved words)
- Comments (single words only, no composite phrases)
- Segment Boundaries (denoted by particular keywords)

4.3.2 General Recognition Process

The input to the indicator recognition process is COBOL II source code.

All indicator recognition methods have the same general structure regardless of the class of indicator that they recognise. There are differences in the matching algorithms for each class to deal with the different types of indicator.

The general structure of indicator recognition is shown in Figure 19.

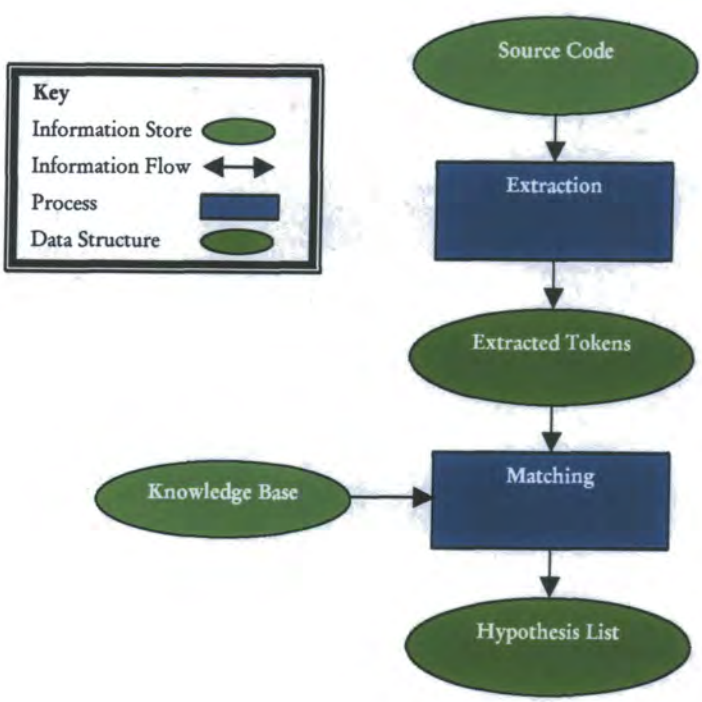


Figure 19: Indicator Recognition Process

4.3.3 Extraction Process

All types of indicator are extracted using a similar process. In each case, a lexical analyser is used to match lexemes belonging to a particular class. A full parser could be employed for more accurate extraction. Only procedure division lexemes are extracted; the reasons for this restriction are given in Chapter 1. It is assumed that the input source code can be compiled and is correct with respect to the language definition. Each lexeme is stored with line and character position information.

Segment boundaries are treated slightly differently. The source code is scanned using a lexical analyser as for the other classes. Discovery of a SECTION lexeme generates a segment-start output with line and character position information. Lexemes EXIT, GOBACK, and STOP generate segment-end output with line and character position information. In the absence of any SECTION lexemes, the PROCEDURE DIVISION lexeme is used to generate a segment-start.

4.3.4 Matching Rules

Once the lexemes have been extracted, they are matched against the indicators in the library to generate hypotheses. In terms of the formal model, the matching test is made between the lexeme string and the data string of an indicator, as shown in the function Match:

Match: (**Indicator, Lexeme**) $\rightarrow$ **Boolean** (22)

Match((*n* : **String**, *c* : **Class**, *d* : **String**), (*s* : **Integer**, *e* : **Integer**, *t* : **String**)) = $d \mu t$

μ (**String, String**) $\rightarrow$ **Boolean** (23)

μ (*d* : **String**, *t* : **String**) = True, if $d = t$ under conditions specified for active options.

The way in which lexemes (also termed tokens) are matched varies, governed by a number of options for each class.

Options available for the classes are:

Identifier	Case Sensitivity Sub-string Matching Synonym Matching
Keyword	No Options
Comment	Case Sensitivity Synonym Matching
Segment Boundary	No Options

Case sensitivity provides greater flexibility when matching strings, particularly in comments where mixed case type is often employed.

Sub-string matching also allows greater flexibility than direct matching because variations of words can be found.

Synonym matching is designed to allow for different words referring to the same concept and requires the availability of a list of common synonyms.

The options may be used in combination as described below, although the more flexible the recognition, the greater the chance of erroneously generating hypotheses.

When a hypothesis is generated, the following information is output:

- A concept
- An indicator
- A lexeme
- Lexeme line number
- Lexeme character position

4.3.4.1 Identifier Matching

If the case sensitivity option is active then make all matches case sensitive, otherwise make all matches case insensitive.

For each lexeme extracted:

1) attempt to match the current lexeme exactly with the tokens stored in the data attribute of every identifier-class indicator in the library. If a match is found, output a hypothesis for each concept in the library that is related to the current library indicator, filling the fields appropriately.

2) if the sub-string matching option is active then attempt to match the current lexeme with the tokens stored in the data attribute of every identifier-class indicator in the library. A match is found if the extracted lexeme is a sub-string of the library data token, or if the library data token is a sub-string of the extracted lexeme. If a match is found, output a hypothesis for each concept in the library that is related to the current library indicator. Hypotheses are not output by this stage if they have already been generated in the exact matching stage described in 1 above.

3) if the synonym matching option is active then attempt to match the current lexeme with the tokens stored in the data attribute of every identifier-class indicator in the library. To determine whether a match has been found, look up synonyms for the current lexeme in the synonym list. For each retrieved synonym, compare it with every library data token in the identifier class. A match is found if the synonym and library tokens are exactly the same, subject to the case sensitivity option. If a match is found, output a hypothesis for each concept in the library that is related to the current library indicator. Hypotheses should not be output by this stage if they have already been generated in either of the two previous stages.

4.3.4.2 Keyword Matching

For each lexeme extracted:

1) attempt to match the current lexeme exactly with the tokens stored in the data attribute of every keyword-class indicator in the library. If a match is found, output a hypothesis for each concept in the library that is related to the current library indicator.

4.3.4.3 Comment Matching

If the case sensitivity option is active then make all matches case sensitive, otherwise make all matches case insensitive.

For each lexeme extracted:

1) attempt to match the current lexeme exactly with the tokens stored in the data attribute of every comment-class indicator in the library. If a match is found, output a hypothesis for each concept in the library that is related to the current library indicator.

2) if the synonym matching option is active then attempt to match the current lexeme with the tokens stored in the data attribute of every comment-class indicator in the library. To determine whether a match has been found, look up synonyms for the current token in the synonym list. For each retrieved synonym, compare it with every library data token in the comment class. A match is found if the synonym and library tokens are exactly the same, subject to the case sensitivity option. If a match is found, output a hypothesis for each concept in the library that is related to the current library indicator. Hypotheses should not be output by the synonym matching stage if they have already been generated in the previous stage.

4.3.4.4 Segment Boundary Matching

No flexible matching criteria are applied in segment boundary matching; each extracted token in the class is output as a boundary hypothesis. If SECTION is found, generate a segment-start hypothesis with line and character position information. If EXIT, GOBACK, or STOP is found, generate segment-end output with line and character

position information. In the absence of any SECTION lexemes, generate a segment-start hypothesis from the PROCEDURE DIVISION lexeme.

4.3.4.5 Output

The output of the indicator recognition process is a hypothesis list. The hypotheses are sorted (if required) into ascending order by line and character position of the generating indicator. There is no specific order on multiple hypotheses from a single indicator.

4.4 Characteristics of Hypothesis Generation

This section compares HB-CA's hypothesis generation process with the equivalent parts of IRENE and DM-TAO. The specific areas of comparison are the initial information source used to begin the concept search, and the type of the initial information.

HB-CA begins its search using source code indicators as discussed above. The initial information source is therefore the source code.

IRENE's initial information is provided in the form of a user-supplied hypothesis. The system proceeds to derive further plausible hypotheses and attempts to find their implementation in the source code being analysed.

DM-TAO in conceptual *grep* mode has a user-supplied concept for which implementations are found. The source-code features used in all cases are syntactic, lexical, and clustering clues [BIGG94]:

Table 6 summarises these differences.

	HB-CA	DM-TAO (Conceptual *grep*)	DM-TAO (Conceptual Highlights)	DM-TAO (Identification)	IRENE
Initial Information Source	Source Code	User	Source Code	Source Code	User
Initial Information	Textual Indicators	User-Supplied Concept, Syntactic, Lexical, and Clustering Clues	Syntactic, Lexical, and Clustering Clues	Syntactic, Lexical, and Clustering Clues	User-Supplied Hypothesis

Table 6: Characteristics of Concept Assignment Methods - Initial Information Sources

4.4.1 Discussion

The advantages of using source code as the primary information source are that it requires no user involvement, and that it causes the search to be focussed on those areas of the knowledge base that are relevant. The disadvantage is that some inferences between concepts may be more difficult to make, and indicators missing from the knowledge base can have a large effect on recognition performance.

Chapter 3 referred to the overproduction of hypotheses during this stage. The problem was discovered during development of the HB-CA method. The technique in question used a knowledge base structure where conceptually similar concepts reinforced each other. This approach was termed *secondary hypothesis*. Whilst in principle this appeared to be a useful idea, it was not successful because hypotheses that should have reinforced each other (e.g. specialised versions of particular hypotheses) actually competed. This was one factor that led to the design of the knowledge base and disambiguation rules.

4.5 Example of Hypothesis Generation

This section demonstrates the application of the hypothesis generation method to the example source code and library content presented in Chapter 3.

For brevity, the entire fragment has not been included here but representative samples are used. Figure 20 shows part of the example code fragment with indicators in the four classes highlighted.

```
GB21    C00-READ-APS SECTION.                                              0193
GB21    C00-000.                                                           0194
GB21  *     READ APS MASTER FILE                                           0195
GB21        CALL 'GBAAY0X' USING APS-RECORD-IN.                            0196
GB21        IF APS-EOF = END-OF-FILE                                       0197
GB21                MOVE HIGH-VALUES TO APS-RECORD-IN                      0198
GB21                GO TO C00-999.                                         0199
GB21        MOVE '1' TO W-GBCM0133-2.                                      0200
GB21        CALL 'GBCM0133' USING APS-RECORD-IN W-GBCM0133-2.              0201
GB21    C00-999.                                                           0202
GB21        EXIT.                                                          0203
GB21        SKIP3                                                          0204
GB21    C10-WRITE-APS SECTION.                                             0205
GB21  *     WRITE APS MASTER FILE                                          0206
GB21        MOVE '2' TO W-GBCM0133-2.                                      0207
GB21        CALL 'GBCM0133' USING APS-RECORD-OUT W-GBCM0133-2.             0208
GB21        CALL 'GBAAZ0X' USING APS-RECORD-OUT.                           0209
GB21    C10-999.                                                           0210
GB21        EXIT.                                                          0211
GB21        SKIP3                                                          0212

Key: XXX - Identifier, XXX - Keyword, XXX - Comment, XXX - Segment Boundary
```

Figure 20: Code Fragment Showing Tokens Classified for Extraction

All of these lexemes would be found by the various indicator extraction methods. Once extracted, matching takes place against the library and Figure 21 shows those indicators that would be found in the example. Active options are: case insensitivity on all modules that support it, and sub-string matching for identifiers.

```
GB21    C00-READ-APS SECTION.                                                    0193
GB21    C00-000.                                                                 0194
GB21  *     READ APS MASTER FILE                                                 0195
GB21        CALL 'GBAAY0X' USING APS-RECORD-IN.                                  0196
GB21        IF APS-EOF = END-OF-FILE                                             0197
GB21                MOVE HIGH-VALUES TO APS-RECORD-IN                            0198
GB21                GO TO C00-999.                                               0199
GB21        MOVE '1' TO W-GBCM0133-2.                                            0200
GB21        CALL 'GBCM0133' USING APS-RECORD-IN W-GBCM0133-2.                    0201
GB21    C00-999.                                                                 0202
GB21        EXIT.                                                                0203
GB21        SKIP3                                                                0204
GB21    C10-WRITE-APS SECTION.                                                   0205
GB21  *     WRITE APS MASTER FILE                                                0206
GB21        MOVE '2' TO W-GBCM0133-2.                                            0207
GB21        CALL 'GBCM0133' USING APS-RECORD-OUT W-GBCM0133-2.                   0208
GB21        CALL 'GBAAZ0X' USING APS-RECORD-OUT.                                 0209
GB21    C10-999.                                                                 0210
GB21        EXIT.                                                                0211
GB21        SKIP3                                                                0212

Key: XXX - Identifier, XXX - Keyword, XXX - Comment, XXX - Segment Break
```

Figure 21: Code Fragment Showing Classified Matched Tokens

The matched indicators produce hypotheses by the methods described above. The first matching token (C00-READ-APS) indicates two concepts: Read and APSRecord. The matching process creates hypotheses for these and stores them in the hypothesis list.

The output can be expressed in terms of the formal model (character positions are representative only) and an extract is shown below:

HL : **Hypothesis List**

HL = {((NRead, Identifier, READ), (Read, Action, Primary), (8025, 8037, C00-READ-APS)), ((NAPS, Identifier, APS), (APSRecord, Object, Secondary), (8025, 8037, C00-READ-APS))...}

4.6 Summary

Chapter 4 has presented the first stage of the HB-CA method, hypothesis generation, describing the key process of indicator recognition. A comparison has been made with the primary knowledge types and sources used by the IRENE and DM-TAO systems. The chapter shows the results of applying hypothesis generation to the example source code and library content given in Chapter 3.

Chapter 5 discusses the next stage of the HB-CA method: segmentation. The problems associated with segmenting programs are presented and a solution based on conceptual clustering is described.

Chapter 5
Segmentation

5.1 Introduction

Chapter 4 presented the first stage of HB-CA, hypothesis generation, which transforms source code into a hypothesis list using the knowledge base. The comprehension activity framework and formal model were extended to show the representations and context of hypothesis generation.

This chapter describes the second stage of HB-CA: segmentation. It is the first major research problem to be addressed by the HB-CA method and involves breaking up the hypothesis list into conceptually coherent segments. The solution clusters concepts in the hypothesis list using segment boundary indicators and self-organising maps (SOMs). The result of segmentation is shown in the context of the comprehension activity framework and appropriate formal representations are defined. Segmentation accepts a hypothesis list as input and produces a hypothesis segment list as output.

5.2 The Segmentation Problem

Segmentation is the problem of determining the location and extent of concepts within a piece of source code, to form segments that then can be labelled. It is a difficult problem because the boundaries between concepts can be confused and fuzzy to the point where two concepts may interleave. Interleaving has been addressed in algorithmic understanders using data and control flow information (see [RUGA96]). It presents a more difficult problem to plausible reasoning understanders, such as HB-CA, where this kind of information is not used. Figure 22 shows an example fragment of source code with two clearly separated concepts.

```
MOVE 'EXAMPLE' TO PRINT-LL.
MOVE '13' TO PRINT-CC.
CALL 'PRINT' USING P-PRINTLINE.
MOVE POLICY-NUM TO OUT-PNUM.
MOVE SCHEME-REF TO OUT-SREF.
CALL 'WRITE' USING OUT-REC.
```

Figure 22: Example Code Fragment Showing Separated Concepts

The first three lines indicate a Print concept; the last three indicate Write. In this situation, it is clear where the boundary between concepts falls. Figure 23 shows the same code but with the boundaries slightly blurred.

```
MOVE 'EXAMPLE' TO PRINT-LL.
MOVE '13' TO PRINT-CC.
MOVE POLICY-NUM TO OUT-PNUM.
CALL 'PRINT' USING P-PRINTLINE.
MOVE SCHEME-REF TO OUT-SREF.
CALL 'WRITE' USING OUT-REC.
```

Figure 23: Example Code Fragment Showing Slightly Merged Concepts

There are still two distinct areas of conceptual focus although the boundary between them is now fuzzy. The final version of this example, shown in Figure 24, demonstrates the concepts when completely merged.

```
MOVE 'EXAMPLE' TO PRINT-LL.
MOVE POLICY-NUM TO OUT-PNUM.
MOVE '13' TO PRINT-CC.
MOVE SCHEME-REF TO OUT-SREF.
CALL 'PRINT' USING P-PRINTLINE.
CALL 'WRITE' USING OUT-REC.
```

Figure 24: Example Code Fragment Showing Completely Merged Concepts

It is now impossible to tell where one concept ends and the other begins. This is confusing in itself, but the confusion is compounded by the fact that there is now a third concept emerging; it could be argued that the last two lines now indicate Call.

It is clear that segmenting even a trivial program is difficult. The problem is considerably greater when addressing real-world heavily-maintained code.

It is possible to perform initial segmentation of a program based on the subroutine structure of the code. This implies an assumption of one concept per subroutine (per section in the case of COBOL II). Although this provides a good starting point, much existing code is poorly structured and may have large subroutines (if they exist at all). Flexible methods are required to detect areas of conceptual focus within subroutines, i.e. those areas where the evidence in the code strongly indicates a particular concept.

The input to the segmentation stage is the hypothesis list generated by the methods described in Chapter 4. The output of the stage is a *hypothesis segment list (HSL).* The HSL can be expressed formally:

$$\textbf{HSL} : \{\textbf{Hypothesis List}\} \tag{24}$$

Segmentation can be seen as a function mapping a hypothesis list to a hypothesis segment list.

$$\textbf{Segmentation} : \textbf{Hypothesis List} \rightarrow \textbf{HSL} \tag{25}$$

Segmentation is the second stage of HB-CA and Figure 25 shows its position in the comprehension activity framework. The hypothesis segment list it produces is indicated by the white oval.

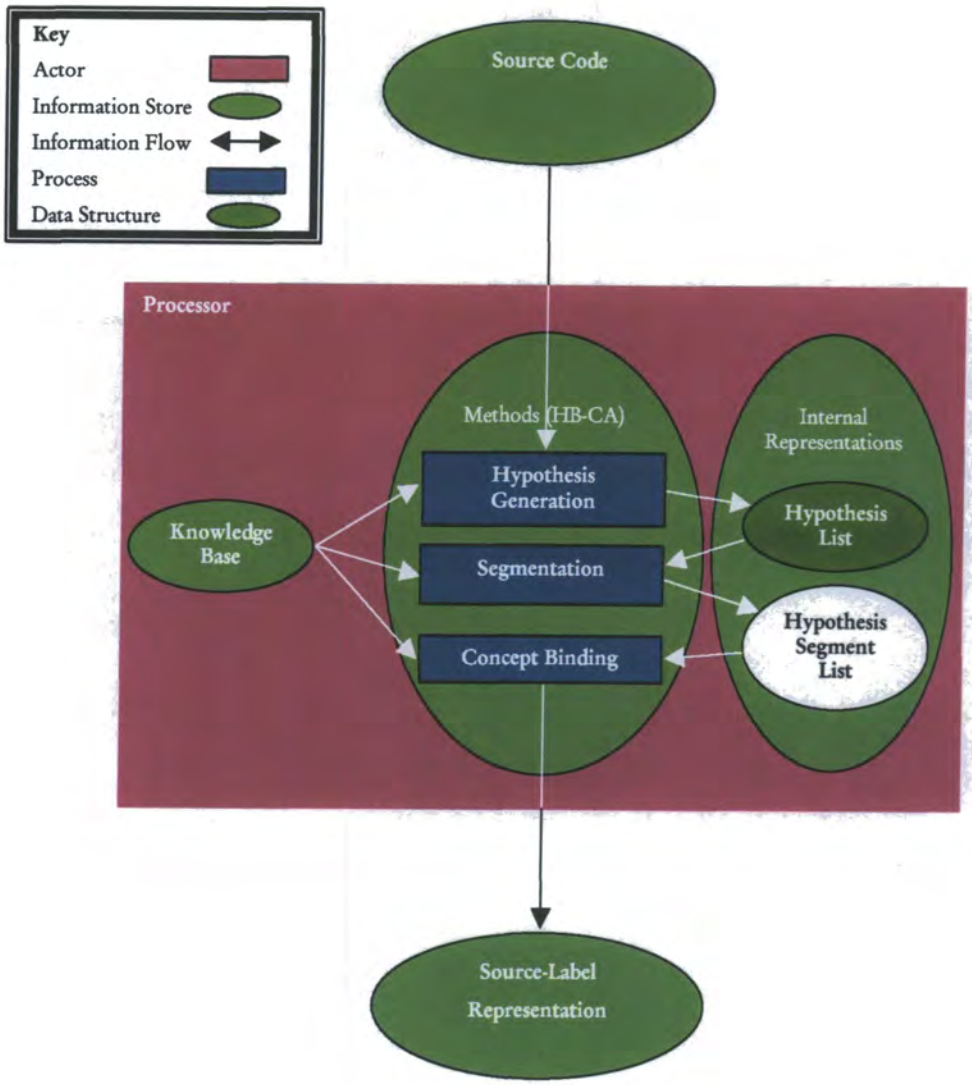


Figure 25: Comprehension Activity Framework Showing the Position of the Hypothesis Segment List

5.3 HB-CA Segmentation

HB-CA adopts a two-stage approach to segmenting the hypothesis list.

5.3.1 Segment Boundary Hypotheses

The first stage assumes that at least one concept should be assigned to every COBOL II section, providing an initial segmentation of the hypothesis list using the segment boundary hypotheses described in Chapter 4. Note that some correction of a segment boundary's position in the list may be required to ensure that all

relevant indicators are included. Figure 26 shows a small example of this situation. The line of code shown, when combined with a knowledge base, produces a hypothesis list, of which a fragment is presented. If the segment boundary is used without correction, both FILE and WRITE will be ignored since hypotheses in the list are considered in the order in which they occur in the source code. The correction algorithm moves the segment boundary hypothesis until it is the first hypothesis occurring on the line being considered. Segment-end hypotheses should not need any correction because they are unlikely to occur on the same line as another token.

```
FILE-WRITE SECTION.
```

HL = {(FILE...),(WRITE...),(SEGSTART)...}

Figure 26: Example Showing Necessity of Boundary Correction

Having established the initial segmentation, further analysis may be required to determine whether these segments can be subdivided to give a greater level of detail about the concepts in the program. Subdivision may be necessary to retain an appropriate level of abstraction for the amount of code being considered. If monolithic code or very large subroutines are being analysed, it is more useful to assign several concepts to parts of each routine than to apply the rule of one concept per subroutine. If a large subroutine is described by one concept, the concept's level of abstraction may need to be raised to accurately represent the operations performed in the routine.

5.3.2 Clustering

HB-CA's method for subdividing segments is based on the idea of finding conceptual clusters within a segment's hypotheses, in other words, to determine areas of strong conceptual focus within the hypothesis list. Applying such a technique to the entire hypothesis list appears attractive but during the development of HB-CA it was found that this caused "unnatural" segmentation. Concept clusters could be formed across subroutine boundaries such that the syntactic structure of the program was not reflected in the concept list. This problem was the

motivation for the initial segmentation algorithm described above, which preserves the syntactic structure of the program.

Early versions of HB-CA attempted clustering using a horizon effect based on the distance, in lines, to the next indicator in the source code. This had the unfortunate effect of occasionally isolating one or two indicators at the end of a subroutine and either ignoring, or misinterpreting the evidence they provided. Moving to a purely hypothesis-based representation, where distances between indicators are determined only by their relative position in the hypothesis list, has helped to eliminate this problem.

5.3.2.1 Pre-Processing

In order to avoid unnecessary work and to derive certain parameters required to perform further clustering if required, each segment's hypotheses are pre-processed according to the following method:

1) For each action-concept hypothesis in a segment, find the concept's most general form by recursively traversing the specialisation relationship in the library. Store the result in a list *F*.
2) If the number of elements in *F* is greater than some user-specified recognition threshold, *rec_thresh*, then continue; otherwise reject the segment and repeat from 1 for the next segment.
3) With a user-specified minimum density for a concept cluster, *min_vd*, determine the number of potential clusters in *F* by dividing the number of elements of *F* by *min_vd*. If the result ≥ 2 then continue, otherwise store this segment in the hypothesis segment list using its initial segment boundaries and repeat from 1 for the next segment.
4) Determine the number of different concepts in *F*. If there is more than 1 then continue, otherwise store this segment in the hypothesis segment list using its initial boundaries and repeat from 1 for the next segment.
5) If this step has been reached, further clustering using self-organising map analysis is required.

The rationale for these steps is now discussed.

All processing at this point is undertaken on action-concept hypotheses only. This reflects the general emphasis on discovering what a program does rather than the objects on which it operates.

Step 1 ensures that versions of the same hypothesis do not compete with each other. If this is not performed, it is possible that the evidence for a particular general concept could be shared among its specialised versions, thus allowing a less strongly indicated concept to win. By finding the most general form of all concepts, comparisons are made at the highest level with evidence for specialisations being used to improve the quality of information later in the process. It should be noted that the other stages of the HB-CA method do not support specialised action-concept hypotheses. Consequently, this step is redundant at present but is included for completeness in the event that HB-CA is extended. Any extension should ensure that the original, specialised hypothesis is replaced in the correct position before concept binding begins.

Step 2 ensures that there are sufficient pieces of evidence for recognition to take place. The user specifies the amount of evidence required.

Step 3 determines the number of potential clusters in *F*. This information is needed to decide whether it is worth attempting to find clusters in the hypotheses. The user specifies the minimum number of hypotheses for a cluster. Dividing the number of hypotheses in *F* by this number gives the maximum number of clusters that could be formed *if* the hypotheses were perfectly clustered initially, a situation unlikely to occur in practice. If there is potential for no more than one cluster then there is no gain from further analysis and the segment can be stored using its current boundaries.

Step 4 ensures that the concepts in *F* are not all the same. If they are all the same then it is clear that the concept to be bound to the segment will be some version of the concepts in *F*, hence there is little point in continued analysis of *F*.

If further analysis and clustering are required, a self-organising map (SOM) is used to find clusters in *F*.

5.3.2.2 Self-Organising Maps (SOMs)

The Self-Organising Map (SOM) (also called a Kohonen network) is an artificial neural network algorithm. It employs unsupervised, competitive learning to perform a topological mapping of high-dimensional input data to a low-dimensional output space. A detailed presentation of the mathematical foundations, variations on the basic algorithm, and an extensive literature survey can be found in [KOHO97]. This section introduces the concepts underlying the SOM and describes the basic algorithm used by HB-CA.

The SOM algorithm performs a vector quantization process, allowing the network to store data whilst maintaining spatial or topological relationships in the training data set, and representing them in a meaningful way [BEAL92]. This is performed by iteratively presenting a set of training vectors to the network and modifying a set of reference vectors to represent those training vectors as accurately as possible. Figure 27 shows the topology of a self-organising map.

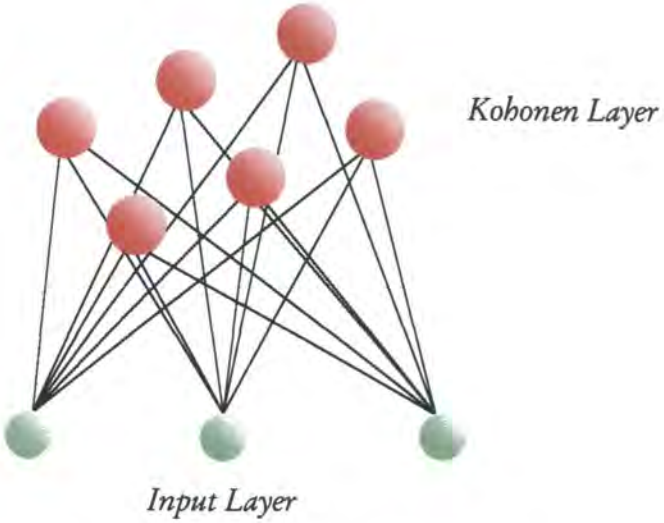


Figure 27: Example of a Self-Organising Map

SOMs have a two layer topology with an input layer the same size as the number of components in the input vectors, and an output layer usually in the shape of a two dimensional grid. Each output node has the same number of vector components as input nodes [ROUS98]. Every input node is connected to every output node. The output node vectors are initialised with random numbers. Learning takes place through the repeated presentation of training data vectors. There may be hundreds

to thousands of repetitions. When a training vector is presented, the Euclidean distance between the training vector and every reference vector stored in the output nodes is calculated. The output node that is closest to the training vector is declared the winner, and its reference vector is updated to reduce its Euclidean distance to the input. In addition, neighbouring nodes in the output layer are also moved proportionally closer to the input. After many repetitions, this process results in the spatial organisation of the input data in clusters of similar, neighbouring regions [ROUS98]. Over the course of training, the size of the neighbourhood and the amount by which Euclidean distances are updated (the learning rate) decrease to zero.

SOMs have many uses including natural language engineering [HONK97], and the organisation of document collections [KASK96].

5.3.2.3 SOMs for HB-CA

The SOM is useful in HB-CA because of its ability to cluster similar data items automatically. Spatial relationships in the segment's hypotheses can be preserved allowing nearby, similar concepts to be clustered together. Consequently, the fuzzy boundaries between areas of conceptual focus in the hypothesis list can be determined using the conceptual content of the list itself, rather than imposing an arbitrary division.

Employing a self-organising map within HB-CA entails solving some additional problems. First, the map must be automatically constructed and the data pre-processed into a vector form. Second, the trained map must be automatically interpreted; a task often left to the user in other SOM applications.

Section 5.3.2.1 described the pre-processing steps. These are designed to ensure that a self-organising map will only be used if there is the potential to form clusters, i.e. the hypothesis list is big enough with a sufficient number of different concepts.

To use the list F with a self-organising map, it must first be turned into a vector representation. A coding scheme must be devised whereby different concepts can be represented as vectors without implying any spatial relationship between them in

a single dimension. It is not possible (or sensible) to represent Read as 1, Print as 2, and Update as 3 in the same dimension, since the ordering relation on integers does not hold for concepts. The solution to this problem arises from SOM work in natural language engineering and document classification. Both [MERK97] and [HONK97] suggest the use of binary vector components to represent categorical data such as the hypotheses in HB-CA. Honkela notes that with large numbers of categories the dimensionality of the vectors would be extremely high [HONK97]. This is not expected to be a problem for HB-CA because the knowledge base is reasonably small. Using binary vector components the concepts Read, Print, and Update, would be placed in different dimensions. A value of 1 in the appropriate dimension would be used to signify the presence of a hypothesis for that concept, 0 would be used otherwise. Given the hypothesis list:

Read, Read, Print, Read, Update, Read

The vector representation would be:

Read	Print	Update
1	0	0
1	0	0
0	1	0
1	0	0
0	0	1
1	0	0

Whilst this would be sufficient input for a self-organising map, HB-CA requires the addition of a further dimension. The data presented above may result in clustering of similar concepts on the SOM. However, this would be meaningless to HB-CA since the map would simply create three clusters, one for each concept. The additional vector component is a sequence number to preserve the order of the hypotheses. This creates a spatial relationship between them, ensuring that clusters will form where the bulk of local evidence for a particular concept occurs (locality is defined in terms of sequence number).

The final vector representation would be:

Seq.	Read	Print	Update
1	1	0	0
2	1	0	0
3	0	1	0
4	1	0	0
5	0	0	1
6	1	0	0

The action concepts in *F* are processed in this manner for use with a SOM.

Having established the data encoding, the map itself must be defined. The documentation for the SOM ToolBox (an implementation of SOM algorithms for Matlab, provided by Kohonen's group) suggests that the number of output neurons should be as large as possible [SOMT00]. For smooth mapping and visual inspection of the output this would be ideal, as clusters would be clearly visible and the mapping could be subtle. The task of the SOM in HB-CA is to cluster hypotheses to enable *automatic* inspection of the output. Consequently, the number of output neurons should be no more than necessary. This creates a coarser granularity in the output space than might be used for visually inspected maps, but forces hypotheses into one of a few groups thus providing sufficient vector density at each neuron for it to be recognised as a cluster. The literature on SOMs does not indicate the widespread use or existence of an algorithmic method to determine the optimal size or shape of a map before training; indeed research is devoted to methods for growing the map to fit the input data during training (see [KOHO97]).

HB-CA addresses the map-sizing problem during the pre-processing phase described in section 5.3.2.1, where the maximum number of clusters is determined.

Assuming a perfectly clustered input list:

Read, Read, Read, Print, Print, Print, Update, Update, Update

and a minimum vector density per cluster of 3, the maximum number of achievable clusters is 3. If the list is less than perfectly clustered, the number of achieved

clusters will be 3 or less since the best case (perfect clustering on input) cannot achieve more. Each output node in the map represents one cluster (once trained, it will trigger for several input vectors) and therefore in this example, the output layer would contain 3 nodes.

A problem for this method can be illustrated by examining what might be considered a worst-case scenario. Assume an input list of the form:

Read, Write, Read, Write, Read, Write, Read, Write, Read, Write

This data is ambiguous since it could be described as having no dominant concept (and hence no clustering). Alternatively, it could be split in half (two output nodes), the first half being dominated by Read and the second by Write. With still more subdivision possible it is hard to say how the data should be clustered, or to determine a suitable size for the output layer using the analysis method suggested. This seems to be an intractable problem for this type of input but since such an even distribution of hypotheses is unlikely to occur often, the method based on perfect clustering is considered suitable for use in all cases.

Having established the number of nodes in the output layer, its shape must also be considered. The most common shape for SOM output layers is a rectangular grid with either a rectangular topology (where nodes update those above, below, left, and right) or a hexagonal topology (where nodes are regarded as having six sides and update those surrounding them accordingly). For the purpose of HB-CA, the output layer is defined as one-dimensional with a rectangular topology. This ensures that the mutual attraction of like hypotheses operates in one dimension only on the map. In theory, a larger two-dimensional map would also work well since the combination of sequence number and concept would ensure that nearby and similar hypotheses group at the same node. Using this type would introduce additional problems, e.g. deciding on the length of each side of the rectangle. This would be particularly difficult if the number of nodes could not be formatted in rectangular fashion. It is tempting to visualise the hypotheses being clustered in sequence from left to right along the output layer although there is no reason why this should happen, especially with random initialisation of the SOM. It should be

noted that in some circumstances a SOM might not be the most efficient approach to clustering. An alternative, such as vector quantization, may be better for situations requiring a 1x2 SOM [NEUR00], but the uniformity of approach outweighs any potential cost saving.

The formatted SOM can now be trained on the input vectors created from *F*. Training for HB-CA takes place in two stages as suggested in [KOHO96]. The first stage orders the reference vectors in the map using a learning rate of 0.05, neighbourhood radius of 1, and neighbourhood type of bubble. Training data is presented 1000 times. The second stage converges the reference vectors on their "correct" values using the same parameters but with a learning rate of 0.02. Data is presented 10000 times.

When training is complete, the map must be interpreted. As SOMs are often applied in data visualisation tasks, it is usual for interpretation to be performed by the user. This is not feasible for HB-CA since the method is fully automatic. HB-CA interprets the SOM by passing the input data through the map once more, taking note of which output node triggers for a particular input vector. Vectors are grouped by the node that they trigger (thus forming a cluster) and are translated back to a hypothesis representation. The particular node triggered by an input vector is not inherently important; it is the association of this input vector with others triggering the same output node that is significant.

The clusters must be analysed to ensure that the required minimum vector density, *min_vd*, is met. Every cluster with $\geq$ *min_vd* vectors (termed a valid cluster) is stored in a list *D*. If every cluster is analysed and *D* remains empty or has one element only, store this segment in the hypothesis segment list with its original boundaries (from segment boundary hypotheses) and begin again with the next one, since zero or one valid clusters have been found.

5.3.2.4 Post-Processing

If *D* has more than one element, further analysis is required. It is possible that, although a number of valid clusters have been found, there are some hypotheses participating in clusters that do not meet the required density. HB-CA takes the

approach of including this information in the valid clusters rather than ignoring it altogether. This ensures that all hypotheses being considered at the start of segmentation are still considered at the end of it. The method used to integrate clusters and hypotheses is naïve, adopting the principle of evenly sharing these items between their surrounding valid clusters. This is performed according to the following steps:

1) Consider the first pair of valid clusters in D, termed A and B. If they are adjacent, in terms of hypotheses, then begin again moving one cluster along such that $A_{new} = B_{old}$.
2) Non-adjacent valid clusters must, by definition, have intervening invalid clusters. Determine the number of intervening invalid clusters, z. If z is an even number, allocate the first $z/2$ invalid clusters to A, and the second $z/2$ invalid clusters to B. Move the start and end points of A and B as necessary to include the additional clusters. If z is odd then allocate $(z-1)/2$ invalid clusters to A and B on their respective sides as for even values of z. The remaining central cluster is divided into its constituent hypotheses. If there is an even number of hypotheses, allocate them equally to A and B (as for clusters); otherwise allocate all but the central hypothesis in this manner. The remaining hypothesis is attached to the largest cluster (or B if the resulting clusters are the same size).
3) Repeat from 1 until there are no more valid clusters to consider.

This method for redistributing hypotheses among areas of strong conceptual focus ensures that no evidence from the hypothesis list is ignored. It can cause problems by producing "loose" segmentation (where a large part of the segment is irrelevant to the concept) and confusing the concept binding process with conflicting evidence. This is an area identified for further work. Despite these potential difficulties, in practice they do not affect the method's performance to a great extent. The clusters formed in D are the basis for new segments in the program.

It is important to recall that all of the work undertaken so far in segmentation has been based on action-concept hypotheses only. Before beginning the concept binding stage, object-concept hypotheses must be reintegrated with the segments.

This is trivial for those segments that have not undergone any subdivision, but for those that have been analysed using the SOM, object-concept hypotheses must be distributed fairly among the new segments. The approach taken is similar to that used above for allocating invalid clusters.

1) Recall that D is a list of clusters returned from the SOM where invalid clusters have been integrated.
2) Move the start of the first element of D to the start-boundary hypothesis for the original segment being considered. Move the end of the last element of D to the end-boundary hypothesis for the original segment being considered. This captures those object-concept hypotheses occurring before the first, and after the last valid cluster.
3) Reintegrate object-concept hypotheses from the hypothesis list that fall within the boundaries of clusters in D. This can be accomplished without difficulty, as it is clear to which cluster the objects belong. Object-concept hypotheses that do not fall within such boundaries are redistributed using the method in 4.
4) Work pair-wise through the clusters in D, analysing object-concept hypotheses in the hypothesis list that fall between the end of the first, and start of the second cluster in each pair. Distribute any intervening object-concept hypotheses evenly between their surrounding clusters in the manner described above for redistributing invalid clusters. When an odd hypothesis remains, attach it to the largest cluster, or the second of the pair if the neighbouring clusters are the same size.

In similar way to integrating invalid clusters, this process ensures that no evidence is lost during the segmentation process. The result of this analysis should be a list in D of adjacent segments with no intervening hypotheses, beginning at the first hypothesis of the original segment before subdivision, and ending at the last hypothesis of that segment. The new segments in D are now stored in the hypothesis segment list instead of the original segment. Repeat the whole process for the remaining segments.

The result of this process is the hypothesis segment list; a list containing all of the hypotheses from the original hypothesis list, divided into segments.

5.4 Characteristics of Segmentation

This section compares the characteristics of the segmentation methods used by IRENE and DM-TAO with that used in HB-CA. The specific comparison criteria are the clustering method, and the data used.

HB-CA's clustering method uses structural information and self-organising maps to find areas of strong conceptual focus. The data for this is the hypothesis representation generated by the first stage, with clustering taking place using concepts.

IRENE does not perform clustering in the same sense as HB-CA. This is because it finds domain concepts with very specific properties using the relationships stored in the knowledge base. As there is no discussion of clustering techniques in [KARA92], further comparison cannot be made.

The literature on DM-TAO is vague when discussing its clustering methods but it appears to use syntactic features in the program to derive clusters based on feature similarity [BIGG93], [BIGG94].

Table 7 summarises these differences:

	HB-CA	DM-TAO (Conceptual *grep*)	DM-TAO (Conceptual Highlights)	DM-TAO (Identification)	IRENE
Clustering Method	Self-Organising Map	Feature Extractors			Unspecified
Clustering Data Used	Hypotheses	Syntactic Features			Unspecified

Table 7: Characteristics of Concept Assignment Methods - Segmentation

5.4.1 Discussion

It is difficult to discuss the relative merits of the approaches with the small amount of information available on IRENE and DM-TAO. As mentioned in section 5.3.2, the hypothesis approach adopted by HB-CA prevents isolated indicators from being ignored, and ensures that in such situations, the segment includes all relevant lines of source code. Using hypotheses should also reduce the cost of modifying HB-CA for additional languages, as the largest changes would need to be made in the simplest stage: hypothesis generation. DM-TAO appears to be largely aimed at discovering clusters of data declarations and its syntactic approach lends itself readily to this task. It is worth noting that whereas HB-CA explicitly segments before binding concepts, the other methods do not make such a clear distinction between the phases.

5.5 Example of Segmentation

This section demonstrates the operation of the methods described in this chapter applied to the example presented in Chapter 3. Chapter 4 used an extract from the example but the complete program fragment is used henceforth.

For brevity, the initial hypothesis list is shown in Figure 28 without ancillary information such as line and character position. The letter before the concept name represents an (A)ction or an (O)bject.

```
A:Read, O:APSRecord, SEGSTART, A:Read, O:APSRecord, O:Record,
O:APSRecord, O:APSRecord, O:Record, O:APSRecord, O:Record,
O:APSRecord, SEGEND, A:Write, O:APSRecord, SEGSTART, A:Write,
O:APSRecord, O:Record, O:APSRecord, O:Record, O:APSRecord,
SEGEND, A:Print, SEGSTART, A:Print, O:Heading, A:Print,
O:Heading, A:Print, A:Print, SEGEND, A:Print, SEGSTART,
A:Print, SEGEND
```

Figure 28: Hypothesis List before Segmentation

The first action in segmentation is to move the segment boundary hypotheses to the correct place. The result is shown in Figure 29 with the relocated hypotheses in red.

```
SEGSTART, A:Read, O:APSRecord, A:Read, O:APSRecord, O:Record,
O:APSRecord, O:APSRecord, O:Record, O:APSRecord, O:Record,
O:APSRecord, SEGEND, SEGSTART, A:Write, O:APSRecord, A:Write,
O:APSRecord, O:Record, O:APSRecord, O:Record, O:APSRecord,
SEGEND, SEGSTART, A:Print, A:Print, O:Heading, A:Print,
O:Heading, A:Print, A:Print, SEGEND, SEGSTART, A:Print,
A:Print, SEGEND
```

Figure 29: Hypothesis List after Segment Boundary Correction

The next stage is pre-processing, described in section 5.3.2.1. Since this section operates only on action concepts, the object concepts are hidden to improve the overall clarity (see Figure 30).

```
SEGSTART, A:Read, A:Read, SEGEND, SEGSTART, A:Write, A:Write,
SEGEND, SEGSTART, A:Print, A:Print, A:Print, A:Print, A:Print,
SEGEND, SEGSTART, A:Print, A:Print, SEGEND
```

Figure 30: Hypothesis List before Pre-Processing

The steps of pre-processing are undertaken and the results are summarised below. Normally each segment would be treated individually but for brevity they are considered together in this example.

1) For each action concept hypothesis in the segment, find the concept's most general form by recursively traversing the specialisation relationship in the library. Store the resulting concept in a list *F*.

In this case, all concepts are already in their general form so the list is unaffected (see Figure 31).

```
SEGSTART, A:Read, A:Read, SEGEND, SEGSTART, A:Write, A:Write,
SEGEND, SEGSTART, A:Print, A:Print, A:Print, A:Print, A:Print,
SEGEND, SEGSTART, A:Print, A:Print, SEGEND
```

Figure 31: Hypothesis List after Pre-Processing

2) If the number of elements in F is greater than some user-specified recognition threshold, *rec_thresh*, then continue; otherwise reject the segment and repeat from 1 for the next segment.

Let *rec_thresh* = 1. Every segment in the above list has more than one action concept and hence none are rejected. The list is unchanged (see Figure 32).

```
SEGSTART, A:Read, A:Read, SEGEND, SEGSTART, A:Write, A:Write,
SEGEND, SEGSTART, A:Print, A:Print, A:Print, A:Print, A:Print,
SEGEND, SEGSTART, A:Print, A:Print, SEGEND
```

Figure 32: Hypothesis List after Checking Threshold

3) With a user-specified minimum density for a concept cluster, *min_vd*, determine the number of potential clusters in F by dividing the number of elements of F by *min_vd*. If the result ≥ 2 then continue, otherwise store this segment in the hypothesis segment list using its initial segment boundaries and repeat from 1 for the next segment.

Let *min_vd* = 3. None of the segments has six or more action concepts and so all are stored using their boundary hypotheses. The final pre-processing step is not relevant to this example. Examples of programs segmented by a SOM may be found in Chapter 8.

The resulting segment list is unchanged from the step before pre-processing, and can be used for concept binding. The list is shown in Figure 33.

```
SEGSTART, A:Read, O:APSRecord, A:Read, O:APSRecord, O:Record,
O:APSRecord, O:APSRecord, O:Record, O:APSRecord, O:Record,
O:APSRecord, SEGEND, SEGSTART, A:Write, O:APSRecord, A:Write,
O:APSRecord, O:Record, O:APSRecord, O:Record, O:APSRecord,
SEGEND, SEGSTART, A:Print, A:Print, O:Heading, A:Print,
O:Heading, A:Print, A:Print, SEGEND, SEGSTART, A:Print,
A:Print, SEGEND
```

Figure 33: Hypothesis List after Checking Cluster Potential

5.6 Summary

This chapter has presented the method by which HB-CA segments the hypothesis list. It has been placed in the context of the comprehension activity framework and the formal model has been extended to capture some appropriate representations. A comparison has been made with IRENE and DM-TAO and the relative merits of each method discussed. The result of applying the segmentation method to the example source code from Chapter 3 is shown.

Chapter 6 presents the methods used for binding concepts in HB-CA. The underlying principles are discussed and a solution based on semantic network activation described.

Chapter 6
Concept Binding

6.1 Introduction

Chapter 5 presented the second stage of HB-CA, which transforms the hypothesis list (generated in the first stage) into a series of segments. The position and output of the stage in the comprehension activity framework were shown and the formal model extended to capture its representations.

This chapter describes the final stage of HB-CA: concept binding. This is the second major research problem addressed by HB-CA and involves determining the appropriate concept for a segment. HB-CA decides on a concept binding using the weight of source-code evidence and the relationships in the library. The position of concept binding in the comprehension activity framework is presented, and the formal model extended and summarised.

6.2 The Concept Binding Problem

Concept binding is the problem of deciding which concept from the knowledge base should be assigned to a particular segment, using the available evidence. This requires a method that can rank possible concepts by the strength of evidence for them. There are various ways that this can be performed. The simplest approach is to count the number of hypotheses for each concept and pick the one with the most evidence. This forms the basic idea underlying several advanced approaches investigated during the development of HB-CA.

The basic idea suffers from being unable to exploit any of the relationships between concepts. This makes sensible disambiguation of equally high-scoring concepts difficult. To alleviate this problem, an early version of HB-CA used the principle of *secondary hypothesis*, where the existence of one concept signified the existence of another. The signified concept achieved an additional but lower score, and further secondary hypotheses were generated from it. The process was repeated until a

required "depth" had been reached. In principle, this approach seems sensible but in practice, it was found to create confusion and the computational cost was high. The method was extended to use a specialisation relationship like that in the current knowledge base but this did not improve performance significantly. Another disadvantage of these approaches is that there is no differentiation between objects and actions, preventing actions from being favoured over objects.

The development of the knowledge base structure described in Chapter 3, allowed the concept binding method to be redefined to take advantage of the relationships and concept types available.

The input to the concept binding stage is the hypothesis segment list (HSL) generated by the method described in the previous chapter. The output of concept binding is a set of labelled segments.

$$\textbf{Concept Binding: HSL} \rightarrow \{\textbf{Labelled Segment}\} \tag{26}$$

A *labelled segment* is defined as a segment attached to a concept label. The concept label can be the name of a single concept, or the combined names of concepts in a composite.

$$\textbf{Labelled Segment: } (s : \textbf{Segment}, n : \textbf{String}) \tag{27}$$

The set of labelled segments resulting from concept binding has the same type as the desired target representation defined in Chapter 2.

The position of concept binding within the comprehension activity framework is shown by the white box in Figure 34.

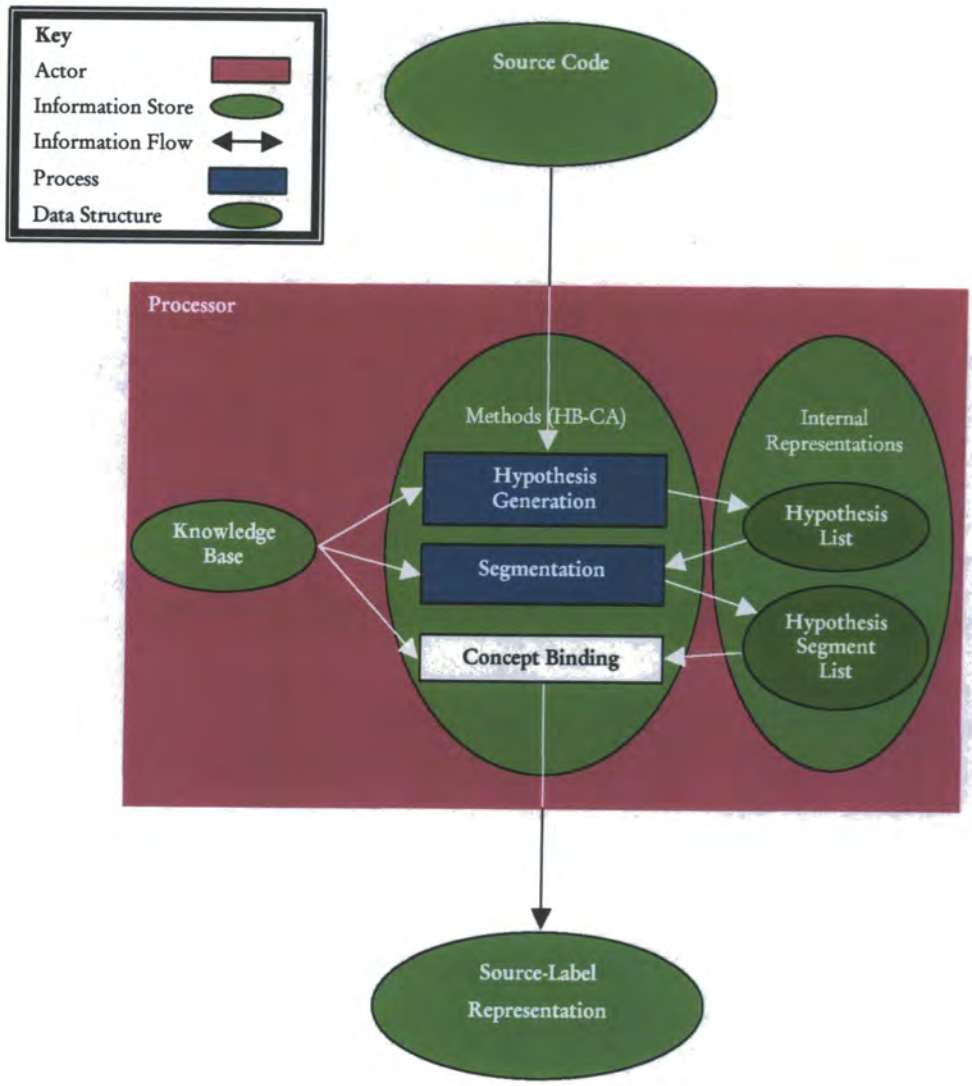


Figure 34: Comprehension Activity Framework Showing the Position of Concept Binding

6.3 HB-CA Concept Binding

HB-CA's concept binding approach uses the basic scoring method described in the previous section as its foundation. It exploits knowledge base relationships and employs a number of disambiguation rules for equally high-scoring concepts. HB-CA concept binding takes the output of the segmentation stage (the hypothesis segment list), scores it, and disambiguates the results to produce a set of labelled segments. The scoring method is introduced in section 6.3.1 in terms of a semantic network. It is then presented again in algorithmic form, with the disambiguation rules shown in section 6.3.2.

6.3.1 Semantic Network "Activation"

Regarding the knowledge base as a semantic network means that the process of assessing evidence can be seen as "activating" parts of the network. The concept with the highest "activation" is considered the winner. The network is "activated" according to the following rules:

- Score 1 for each hypothesised concept.
- Score 1 for the appropriate side of every composite in which this concept participates.
- Score 1 for each more general version of this concept, and the appropriate side of any composite in which the more general version participates.

These rules are designed to bias the scoring towards certain types of concept. The basic principle of winning by weight of evidence is captured in the first rule where hypotheses for a concept increase its score. The second rule ensures that if composites exist (and there is object evidence), they will win in preference to single concepts. The principle is that a composite provides a more informative label for a segment and should win if possible. Giving scores to more general versions of the hypothesised concept (the third rule) has two purposes:

1) To ensure a logical approach to the evidence. If MasterFile has been hypothesised, it is reasonable to say that there is evidence of a File.
2) To manage conflicting evidence. If a general concept has two specialised versions, each with the same score, it is impossible to tell which should win. Scoring the general concept in addition to its specialised versions ensures that it will always score the same or higher than either one. If there is no conflict then the direct evidence for MasterFile should override the indirect evidence for File. Applying this prioritisation can be left to the user's discretion.

The following example illustrates the application of these rules to a simple semantic network.

6.3.1.1 Example of Semantic Network "Activation"

Consider the following hypothesis list:

Print, Read, Record, MasterFile, Read

and the semantic network shown in Figure 35 (indicators are not included).

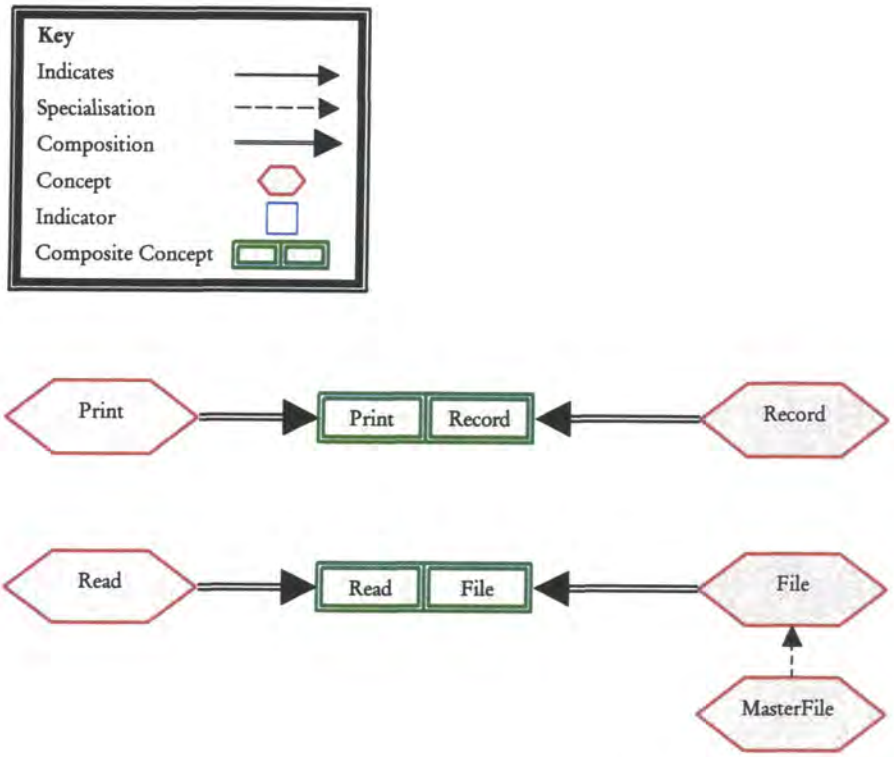


Figure 35: Semantic Network before Scoring

The following series of figures demonstrates the scoring process and the effect this has on the semantic network. Higher scores are indicated by a larger 3D effect on a particular node. Scores for composite nodes are the sum of the action and object concept scores. Figure 36 shows the effect of scoring the first hypothesis.

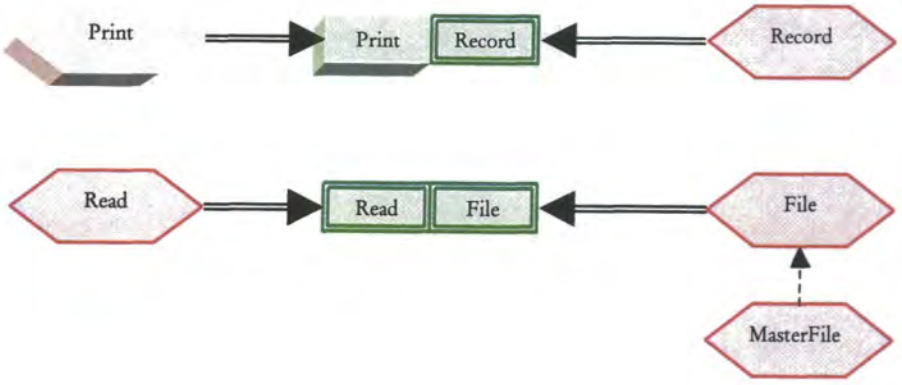


Figure 36: Semantic Network after Scoring Print

The Print node and the composite in which it participates both gain one point. This is repeated for Read in Figure 37.

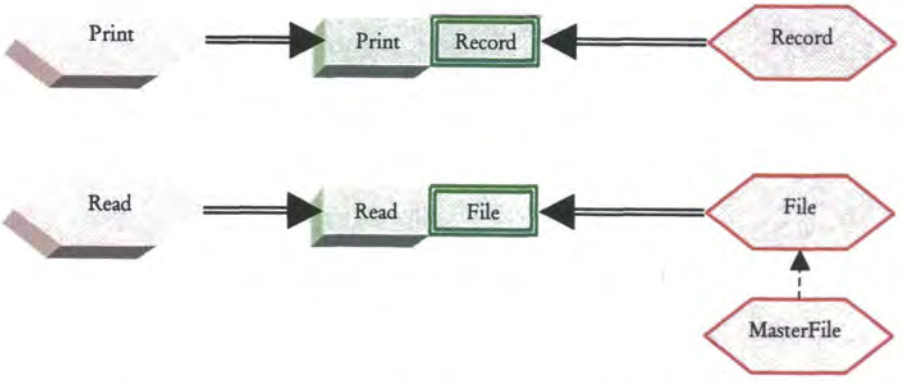


Figure 37: Semantic Network after Scoring Read

Figure 38 shows the network after scoring the Record concept.

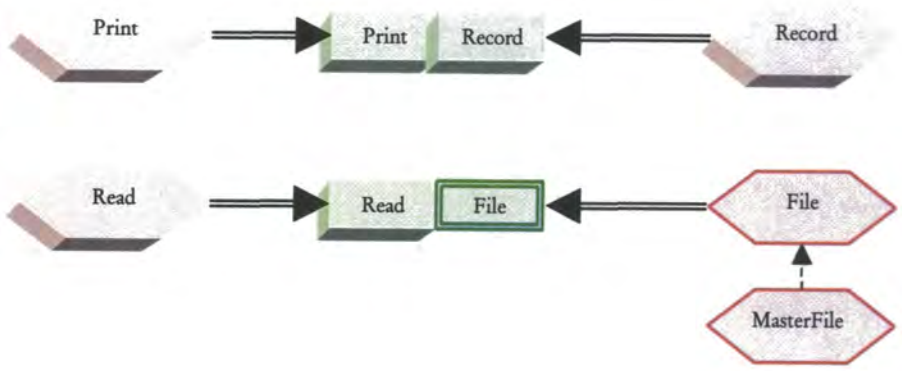


Figure 38: Semantic Network after Scoring Record

Scoring the MasterFile concept has a more significant impact on the network. It is a specialised version of the File concept and consequently this gains a point for being a more general form of the hypothesised concept. Since File now has a point, the composite in which it participates also requires one. This is depicted in Figure 39.

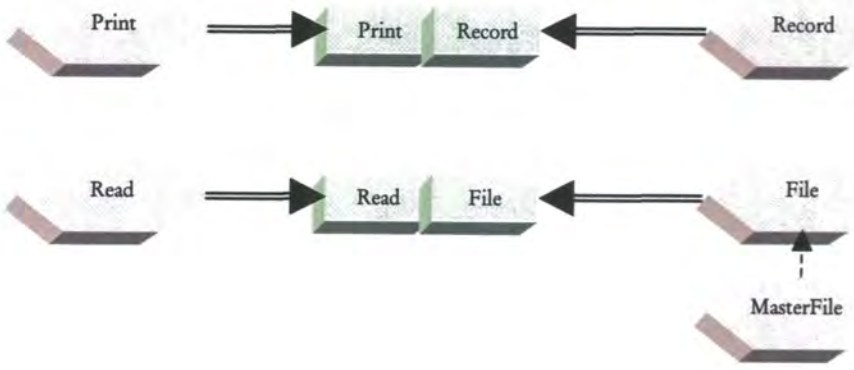


Figure 39: Semantic Network after Scoring MasterFile

The final Read concept is now scored as shown in Figure 40.

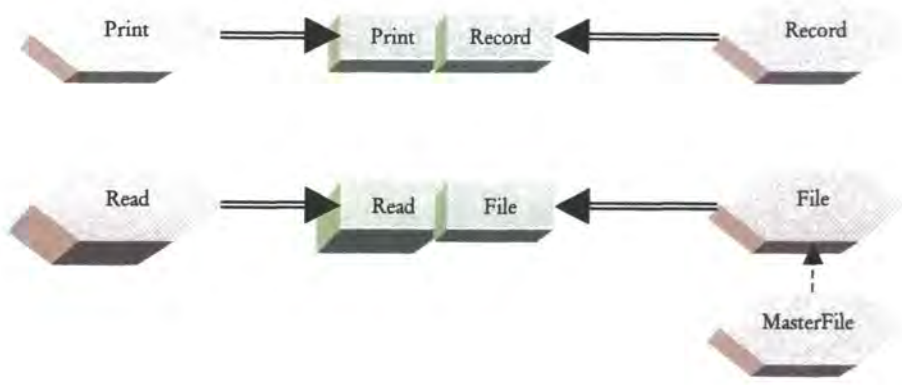


Figure 40: Semantic Network after Scoring Read

The resulting activation levels show a clear winner in Read:File.

6.3.2 Concept Binding Algorithm

The semantic network scoring rules provide a useful way of understanding the general principles of the scoring process. Despite including reasonable bias in the scoring model, they do not address situations where several equally high scoring winners exist; neither do they provide a systematic process. These weaknesses are addressed in this section and the exact algorithm used for concept binding in HB-CA is presented.

Conclusions play a central role in the algorithmic presentation of HB-CA concept binding. The method generates possible conclusions from the current segment's action-concept hypotheses and the library. These conclusions are then reinforced by the segment's object-concept hypotheses, and various rules are applied to select a winner from the result. Action concepts are considered before objects, to reflect the emphasis on determining what is taking place in the program, rather than which objects are involved in the action. Action conclusions can exist without a composite object but the reverse is not true because of HB-CA's aim of determining computational intent.

Conclusions can be composite or non-composite in a similar way to hypotheses. The only difference is that conclusions have an associated score, with composite conclusions having separate action and object scores.

6.3.2.1 Conclusion Generation

The first stage of concept binding is to generate possible conclusions from the action-concept hypotheses. This is undertaken according to the following method:

1) Let *C* be an empty list of conclusions.
2) For the current segment, select all action concept hypotheses and place in a list *AH*.
3) For every element of *AH*:
    a. Let *ac* be the current element of *AH*.
    b. Find all object concepts within the library that participate in a composite concept with *ac*. Store in a list *OC* removing duplicates.
    c. Find all specialisations of all elements of *OC* and add them to *OC*, removing duplicates.
    d. If a non-composite conclusion for *ac* exists in *C*, increase its score by 1, otherwise, store a non-composite conclusion for *ac* and set its score to 1.
    e. Generate composite conclusions in *C* by composing *ac* with every member of *OC* in turn. If a particular composite conclusion already exists, increase its action score by 1; otherwise, store the composite conclusion with its action score set to 1, and its object score set to 0.

The result of this stage is a list *C* of composite and non-composite conclusions.

6.3.2.2 Conclusion Completion and Reinforcement

Object-concept hypotheses are now employed to reinforce and complete composite conclusions.

1) For the current segment, select all object-concept hypotheses and place in a list *OH*.
2) For every element of *OH*:
    a. Let *oc* be the current element of *OH*.
    b. Step through *C* to find the first (or next) composite conclusion, *cc*.
    c. If the object concept assigned to *cc* is *oc* then increase the object score for *cc* by 1.
    d. Repeat from b until no more composites can be found in *C*.
    e. If *oc* is not a primary concept, step back one level along the specialisation relationship and repeat from b, starting with the first element of *C*.

The result of this stage is the list *C* with the same non-composite conclusions as before, but with some composite conclusions now having non-zero scores for both action and object concepts.

6.3.2.3 Disambiguation

This stage applies a number of rules to determine the dominant concept in the current segment. Before beginning to apply them, the list *C* is processed to remove incomplete conclusions. These are composite conclusions with zero object concept score, i.e. there was no evidence for the pairing of the particular action and object.

Having removed incomplete conclusions, let *hs* be the highest score achieved by any conclusion in *C*. The score will be either the non-composite conclusion score or the sum of the action and object scores for composite conclusions.

The following steps are now undertaken:

1) Find the conclusions in *C* that score *hs* using the non-composite conclusion scores, or the sum of the action and object scores for composite conclusions. Store these conclusions in a list *W*.
2) If *W* has more than one element, remove any conclusions from *W* that are specialisations of other conclusions in *W*. This leaves only the most general forms of composite conclusions, and all non-composite conclusions.
3) If *W* still has more than one element, favour composite conclusions over non-composite ones. Remove non-composite conclusions from *W* if there are composite conclusions in *W*.
4) If *W* still has more than one element, find the highest score achieved by the concepts of non-composite conclusions, and the action components of composite conclusions. Remove any non-composite conclusions from *W* that do not score at this level, and any composite conclusions whose action score does not reach this level.
5) If *W* still has more than one element, determine whether the action concepts of the remaining conclusions are the same. If so, then select the non-composite concept (which may or may not be in *W*) corresponding to the action concepts of the conclusions in *W*. Remove all elements of *W* except for this non-composite conclusion. Its score should be increased by the number of elements in the list when rule 5 was invoked. If the action concepts in the remaining conclusions are not the same, the decision must be arbitrary. Remove all but the first conclusion in *W*.

The conclusion remaining in *W* is declared the winner.

6.3.2.4 Post-Disambiguation Processing

The user has the option of forcing the most specialised form of the winning concept to be selected (assuming the winner is composite). Note that specialisations can be selected only if there is evidence for them. This is undertaken by setting the *forced_specialisation* parameter to True. If this is the case, all composite conclusions in *C* that have the same action and a more specialised form of the object concept of the winning composite, are placed in a list *Q*. The highest score

in Q is found and if only one conclusion gains this score, the winner is replaced by the more specific version. If more than one conclusion gains this score, the evidence for a specialised version is ambiguous and the original winner is not replaced.

Finally, if the winner is composite and its combined action and object scores ≥ *rec_thresh* x 2, bind the conclusion to this segment, thus labelling it. If the winner is not composite, bind the conclusion if its score > *rec_thresh*. If neither condition holds, the current segment should be rejected. The difference in threshold between the two types of conclusion forces the evidence required for a non-composite conclusion to be greater than that for a composite conclusion, as the spread of evidence in the composite is regarded as increasing a conclusion's plausibility.

6.3.2.5 Output

The resulting concept label is attached to the current segment and then can be displayed in an appropriate format. The extent of the segment in terms of source code lines can be traced using the hypothesis list and the code position of the indicators that created the first and last hypotheses in the segment.

The concept binding process is carried out for each segment.

6.3.2.6 Discussion

The rationale for the rules described in section 6.3.2.3 is explained here. Generally, the aim is to ensure fair competition between the highest scoring conclusions, whilst maintaining the greatest possible level of detail in the result.

The removal of incomplete conclusions is designed to ensure that only those with evidence for all of their parts are considered. This is not a problem for non-composite conclusions since they cannot be created in the list without evidence. By generating all composites from them, a number of objects may be suggested for use in conclusions without any direct evidence of their existence.

Rule 1 ensures that only the strongest conclusions are considered by the later rules.

Rule 2 aims to ensure that a fair competition is taking place between the conclusions. Specialised versions of a conclusion should not compete with more general versions since they are indicating the same concept at a different level of abstraction. The specialisation can be removed safely as it can be retrieved later if the general concept wins.

Rule 3 reflects the bias built into the scoring algorithm itself, favouring composites over non-composites. The assumption underlying this rule is that a composite conclusion can provide more information than a non-composite one. If they have scored the same, the evidence is more widely distributed for the composite (since the sum of the object and action scores is the same as the score for the non-composite). The larger spread of evidence should ensure a more plausible concept assignment, as both actions and objects indicate the concept. The object evidence is used to "validate" the action conclusion and the probable relationship between them increases the plausibility.

Rule 4 favours actions over objects by considering the scores of non-composites and only the action portion of composites. Thus, higher scoring actions are given priority over lower scoring actions with strong object evidence. This reflects HB-CA's bias towards actions.

Rule 5 checks whether the remaining conclusions are based on the same root action e.g. Read:File, Read:Record, Read:Disk. If this is the case, the evidence for the objects is ambiguous (they must all have the same score to have survived the application of rule 4) and hence the action is left on its own. Note that the single action may no longer exist in the highest scoring conclusion list, having been removed by rule 3. If this is the case, then it is reintroduced to replace the ambiguous composites and is declared the winner. Its score is increased to take account of the multiple conclusions it replaces, and to increase the chance that it will pass the recognition threshold.

If applying all the rules fails to leave only one winner, an arbitrary decision is made.

The disambiguation ability of these rules is discussed further in section 8.4.

6.4 Characteristics of Concept Binding

This section compares the characteristics of the concept binding methods used by IRENE and DM-TAO with that of HB-CA. The specific criteria are: evidence used, assessment method, and explanatory power.

HB-CA uses the evidence provided by the hypothesis segment list in the form of concept hypotheses. The evidence is assessed by scoring each possible conclusion from the library based on the contents of the HSL. Ambiguity is resolved by the application of various rules. The decision made by HB-CA can be explained to a reasonable level of detail since each rule has a particular purpose, hence conclusions can be rejected for a clear reason.

IRENE largely uses evidence from the domain model to bind concepts. A candidate concept is selected and its correspondence to a data name established. The parse tree is searched for possible implementations of related items based on their position in the rule and program syntax, e.g. if *tax* has been related to TAX in the program and a rule is defined as "*taxable_salary - net_salary* derives *tax*", the parse tree might be searched for instances of SUBTRACT y FROM x GIVING TAX. All other rules deriving *tax* also would be sought. Assume the statement SUBTRACT NET FROM GROSS GIVING TAX was found. If considered plausible enough, the two tokens, NET, and GROSS, would be bound to *net_salary* and *taxable_salary* respectively. A similar but more detailed example is shown in [KARA92]. Plausibility is established by summing the weights of the various rules triggered by a particular candidate concept implementation. Lexical matching rules carry a lower weight than domain rules owing to IRENE's emphasis on domain knowledge [KARA92]. The system has good explanatory power and is capable of rewriting the rule-triggering process in English, substituting variable names where appropriate.

DM-TAO uses evidence direct from feature extractors. This is assessed by the semantic connectionist network that forms the heart of the system. Extracted features trigger nodes in the input layer of the network. The signals generated in this layer propagate through the network triggering other types of node. This

continues until concept nodes are triggered and their output level is higher than a given threshold. The nature of this type of network means that DM-TAO cannot easily explain the reasons behind its concept binding.

The approaches to concept binding are summarised in Table 8.

	HB-CA	DM-TAO (Conceptual *grep*)	DM-TAO (Conceptual Highlights)	DM-TAO (Identification)	IRENE
Concept Binding Evidence	Hypotheses	Syntactic Features			Syntactic Features/ Domain Model
Concept Binding Method	Scored Weight of Evidence with Disambiguation Rules	Connectionist Network Triggering and Propagation			Plausibility Measure using Weighted Matching Rules
Explanatory Power	Medium	Low			High

Table 8: Characteristics of Concept Assignment Methods - Concept Binding

6.4.1 Discussion

The concept binding methods discussed here are linked strongly to those used for segmentation. This is not surprising since these two stages of concept assignment are crucial to the success of any particular approach and must work well together. The systems that use rules in their inference (HB-CA and IRENE) are better at explaining their actions. This is balanced by the fact that DM-TAO has a finer-grained inference mechanism that is capable of being updated automatically. The evidence used by the systems for concept binding is largely the same as that used throughout each one for other purposes. The exception is IRENE, which uses syntactic features to a greater extent in concept binding than in other parts of its operation.

6.5 Example of Concept Binding

This section presents the algorithms described in this chapter, applied to the example code and semantic network shown in Chapter 3. The hypothesis segment list produced in Chapter 5 is used as the input to this stage. This is shown again in Figure 41.

```
SEGSTART, A:Read, O:APSRecord, A:Read, O:APSRecord, O:Record,
O:APSRecord, O:APSRecord, O:Record, O:APSRecord, O:Record,
O:APSRecord, SEGEND, SEGSTART, A:Write, O:APSRecord, A:Write,
O:APSRecord, O:Record, O:APSRecord, O:Record, O:APSRecord,
SEGEND, SEGSTART, A:Print, A:Print, O:Heading, A:Print,
O:Heading, A:Print, A:Print, SEGEND, SEGSTART, A:Print,
A:Print, SEGEND
```

Figure 41: Hypothesis Segment List for Concept Binding

The list shows four segments for concept binding and the first contains the following hypotheses:

```
SEGSTART, A:Read, O:APSRecord, A:Read, O:APSRecord, O:Record,
O:APSRecord, O:APSRecord, O:Record, O:APSRecord, O:Record,
O:APSRecord, SEGEND
```

These hypotheses are used in the worked example below.

Concept binding begins with conclusion generation. Action-concept hypotheses are considered first, beginning with Read. This gains a score of 1 for its direct evidence. Composite conclusions based on Read are now generated. In this case there are two: Read:Record, and Read:APSRecord. These gain an action score of 1. The conclusion list after scoring the first action-concept hypothesis is:

Read 1
Read:Record 1:0
Read:APSRecord 1:0

The remaining action-concept hypotheses are now addressed in the same way (in this case there is only one, another Read hypothesis). The conclusion list is now:

Read 2
Read:Record 2:0
Read:APSRecord 2:0

As there are no more action-concept hypotheses, the object-concept hypotheses are considered to reinforce and complete the composite conclusions. The first is APSRecord. This completes the last conclusion in the list, but since it is a specialisation of Record, Read:Record will also be completed. The list is now:

Read 2
Read:Record 2:1
Read:APSRecord 2:1

The next hypothesis is also APSRecord leaving the list as:

Read 2
Read:Record 2:2
Read:APSRecord 2:2

This is followed by a Record hypothesis. Since this is already primary, it only reinforces those conclusions in which it participates.

Read 2
Read:Record 2:3
Read:APSRecord 2:2

The remaining object-concept hypotheses are processed, leaving the list in its final state of:

Read 2
Read:Record 2:9
Read:APSRecord 2:4

There are no incomplete conclusions to remove so the disambiguation stage commences. The highest scoring conclusion is Read:Record and since it is the only one to score 11 in total, it is declared the winner without the need to invoke further disambiguation rules.

If the user requires the most specific version of conclusions to be found, Read:Record would be replaced by Read:APSRecord.

Assuming *rec_thresh* to be the same as in Chapter 5 (i.e. equal to 1), either conclusion would be acceptable for concept binding.

Repeating this process for each segment in the HSL yields the following results (assuming the most specific versions are required):

Segment 1: Read:APSRecord
Segment 2: Write:APSRecord
Segment 3: Print:Heading
Segment 4: Print

These results appear to be correct with respect to the original source code (see Figure 42).

```
GB21    C00-READ-APS SECTION.                                                0193
GB21    C00-000.                                                             0194
GB21  *     READ APS MASTER FILE                                             0195
GB21        CALL 'GBAAY0X' USING APS-RECORD-IN.                              0196
GB21        IF APS-EOF = END-OF-FILE                                         0197
GB21                MOVE HIGH-VALUES TO APS-RECORD-IN                        0198
GB21                GO TO C00-999.                                           0199
GB21        MOVE '1' TO W-GBCM0133-2.                                        0200
GB21        CALL 'GBCM0133' USING APS-RECORD-IN W-GBCM0133-2.                0201
GB21    C00-999.                                                             0202
GB21        EXIT.                                                            0203
GB21        SKIP3                                                            0204
GB21    C10-WRITE-APS SECTION.                                               0205
GB21  *     WRITE APS MASTER FILE                                            0206
GB21        MOVE '2' TO W-GBCM0133-2.                                        0207
GB21        CALL 'GBCM0133' USING APS-RECORD-OUT W-GBCM0133-2.               0208
GB21        CALL 'GBAAZ0X' USING APS-RECORD-OUT.                             0209
GB21    C10-999.                                                             0210
GB21        EXIT.                                                            0211
GB21        SKIP3                                                            0212
GB21    C20-PRINT SECTION.                                                   0213
GB21    C20-000.                                                             0214
GB21  *     PRINT PECULIAR RECORDS TO BE MANUALLY CHECKED                    0215
GB21        IF A-LINENO LESS THAN 25                                         0216
GB21                GO TO C20-010.                                           0217
GB21                                                                         0218
GB21        ADD 1 TO A-PAGENO.                                               0219
GB21        MOVE A-PAGENO TO H1-PAGE.                                        0220
GB21        MOVE C-1 TO P-CC.                                                0221
GB21        MOVE H1-HEADLINE TO P-LL.                                        0222
GB21        PERFORM S00-PRINT.                                               0223
GB21                                                                         0224
GB21        MOVE WS-2 TO P-CC.                                               0225
GB21        MOVE H1-HEADLINE TO P-LL.                                        0226
GB21        PERFORM S00-PRINT.                                               0227
GB21        MOVE 0 TO A-LINENO.                                              0228
GB21                                                                         0229
GB21    C20-010.                                                             0230
GB21        MOVE WS-2 TO P-CC.                                               0231
GB21        MOVE GBAIA010 TO P1-KEY.                                         0232
GB21        MOVE P1-DATALINE TO P-LL.                                        0233
GB21                                                                         0234
GB21        PERFORM S00-PRINT.                                               0235
GB21        MOVE SPACES TO P-LL.                                             0236
GB21        ADD 2 TO A-LINENO.                                               0237
GB21    C20-999.                                                             0238
GB21        EXIT.                                                            0239
GB21        EJECT                                                            0240
GB21    S00-PRINT SECTION.                                                   0241
GB21    S00-000.                                                             0242
GB21  *     PRINTS A LINE                                                    0243
GB21                                                                         0244
GB21        CALL 'PRINT' USING P-PRINTLINE.                                  0245
GB21    S00-999.                                                             0246
GB21        EXIT.                                                            0247

Read:APSRecord          Write:APSRecord              Print:Heading       Print
```

Figure 42: Example Source Code Highlighted to Indicate Labelled Segments

6.6 Summary of Formal Model

A formal model describing the various representations used by HB-CA has been developed throughout this thesis. This section collects all the definitions to summarise the model in a coherent manner.

Chapter 2 introduced the formal model, characterising HB-CA as a way of mapping a source representation to a target representation (definition 8).

$$\textbf{Source} : \{x : \textbf{Line}\} \tag{1}$$

$$\textbf{Line} : (\{y : \textbf{Lexeme}\}, \textit{seqnum} : \textbf{Integer}) \tag{2}$$

$$\textbf{Lexeme} : (\textit{start} : \textbf{Integer}, \textit{end} : \textbf{Integer}, \textit{token} : \textbf{String}) \mid \textit{start} \leq \textit{end} \tag{3}$$

$$\textbf{TR} : \{(x : \textbf{Segment}, y : \textbf{String})\} \tag{4}$$

$$\textbf{Concept} : \textbf{String} \tag{5}$$

$$\varphi : (\textbf{Line}, \textbf{Line}) \rightarrow \textbf{Boolean}$$
$$\varphi\,((a, b), (c, d)) = b \leq d \tag{6}$$

$$\textbf{Segment} : (\textit{start} : \textbf{Line}, \textit{end} : \textbf{Line}) \mid \textit{start}\ \varphi\ \textit{end} \tag{7}$$

$$\textbf{P} : \textbf{Source} \rightarrow \textbf{TR} \tag{8}$$

Chapter 3 extended the model by introducing definitions of the knowledge base (definition 18) and its constituent parts.

$$\textbf{Class} : \textbf{String}$$
$$\forall X : \textbf{Class}, X \in \{\text{"Identifier"}, \text{"Keyword"}, \text{"Comment"}, \text{"Segment Boundary"}\} \tag{9}$$

$$\textbf{Indicator} : (n : \textbf{String}, c : \textbf{Class}, d : \textbf{String}) \tag{10}$$

$$\textbf{Level} : \textbf{String}$$
$$\forall X : \textbf{Level}, X \in \{\text{"Primary"}, \text{"Secondary"}\} \tag{11}$$

$$\textbf{Type} : \textbf{String}$$
$$\forall Y : \textbf{Type}, Y \in \{\text{"Action"}, \text{"Object"}\} \tag{12}$$

$$\textbf{Concept} : (n : \textbf{String}, l : \textbf{Level}, t : \textbf{Type}) \tag{13}$$

$$\textbf{Indicates} : \{(p : \textbf{Indicator}, q : \textbf{Concept})\} \tag{14}$$

$$\text{CCR} : \{r \mid r : \{(a : \textbf{Concept}, b : \textbf{Concept})\}\} \quad (15)$$

$$\textbf{Specialisation} : \{((a,b,c):\textbf{Concept}, (d,e,f):\textbf{Concept}) \mid e = \textit{"Secondary"}\} \quad (16)$$

$$\textbf{Composition}:\{((a,b,c):\textbf{Concept}, (d,e,f):\textbf{Concept}) \mid b = \textit{"Primary"}, e = \textit{"Primary"}, c = \textit{"Action"}, f = \textit{"Object"}\} \quad (17)$$

$$\textbf{KB} : (\{x : \textbf{Concept}\}, \{i : \textbf{Indicator}\}, \{(p : \textbf{Indicator}, q : \textbf{Concept})\}, \{r \mid r : \{(a : \textbf{Concept}, b : \textbf{Concept})\}\}) \quad (18)$$

Recall that definition 13 extends definition 5.

With the key representations defined, Chapter 4 presented the first stage of the HB-CA process: hypothesis generation. This was defined as a function, mapping source to a list of hypotheses (definition 21).

$$\textbf{Hypothesis} : (i : \textbf{Indicator}, c : \textbf{Concept}, l : \textbf{Lexeme}) \quad (19)$$

$$\textbf{Hypothesis List} : \{h : \textbf{Hypothesis}\} \quad (20)$$

$$\textbf{HG} : \textbf{Source} \rightarrow \textbf{Hypothesis List} \quad (21)$$

Recall that the knowledge base has been deliberately omitted from the function definitions here.

Hypotheses are generated using various matching rules.

$$\textbf{Match: (Indicator, Lexeme)} \rightarrow \textbf{Boolean} \quad (22)$$

$$\textbf{Match}((n : \textbf{String}, c : \textbf{Class}, d : \textbf{String}), (s : \textbf{Integer}, e : \textbf{Integer}, t : \textbf{String})) = d \,\mu\, t$$

$$\mu\ (\textbf{String, String}) \rightarrow \textbf{Boolean} \quad (23)$$

$\mu\ (d : \textbf{String}, t : \textbf{String})$ = True, if $d = t$ under conditions specified for active options.

Chapter 5 described the next stage of HB-CA: segmentation. This maps the output of hypothesis generation to a hypothesis segment list by breaking the hypothesis list into groups (definition 25). The formal representation of this is:

$$\textbf{HSL} : \{\textbf{Hypothesis List}\} \tag{24}$$

$$\textbf{Segmentation} : \textbf{Hypothesis List} \rightarrow \textbf{HSL} \tag{25}$$

The final stage of HB-CA, concept binding, was presented in this chapter. It maps the output of segmentation to a collection of labelled segments.

$$\textbf{Concept Binding: HSL} \rightarrow \{\textbf{Labelled Segment}\} \tag{26}$$

$$\textbf{Labelled Segment: } (s : \textbf{Segment}, n : \textbf{String}) \tag{27}$$

This concludes the summary of existing definitions. By comparing definitions 26 and 4, it can be seen that the output of concept binding has the same type as the required target representation.

Definition 8 characterised the original problem in terms of a mapping between source and the required target representation. By creating a composition of the functions that represent each part of HB-CA, it can be shown that HB-CA provides a solution to the original problem.

$$\textbf{P} : \textbf{Source} \rightarrow \textbf{TR} \tag{8}$$

$$\textbf{HB-CA} : \textbf{Source} \rightarrow \{\textbf{Labelled Segment}\} \tag{28}$$

$$\textbf{HB-CA} : \textbf{Concept Binding} \circ \textbf{Segmentation} \circ \textbf{HG} \tag{29}$$

6.7 Summary

This chapter has presented the final stage of HB-CA: concept binding. It has been placed in the context of the comprehension activity framework and formal model. Comparisons have been made with the concept binding methods of IRENE and DM-TAO, and the merits of each discussed. Applying the concept-binding algorithm to the hypothesis segment list shown in Chapter 5 has completed the worked example initiated in Chapter 3. The formal model has been summarised and the HB-CA process characterised as the composition of three functions, each representing a stage of HB-CA.

Chapter 7 describes an implementation of HB-CA called the Hypothesis-Based Concept Assignment System (HB-CAS). Various issues relating to the design and implementation are discussed.

Chapter 7
Implementation

7.1 Introduction

Chapter 6 presented the final stage of HB-CA: concept binding. This completed the description of the HB-CA method by showing the way a hypothesis segment list is transformed into the target representation defined in Chapter 2. The position of concept binding in the comprehension activity framework was shown and the formal model extended and summarised. Concept binding was applied to the worked example initiated in Chapter 3.

This chapter discusses the implementation of HB-CA in the Hypothesis-Based Concept Assignment System (HB-CAS). The design of HB-CAS is presented and various technical issues discussed. A short evaluation of the implementation is presented.

7.2 System Implementation

The methods presented in the preceding chapters are embodied in the HB-CAS system. HB-CAS runs on the Microsoft Windows 95/98 operating system.

7.2.1 Programming Environment

Development was undertaken using Borland Delphi 4.0. This language was chosen for several reasons:

- It supports rapid prototyping of graphical user interfaces.
- The underlying source language (Object Pascal) is stable and well defined.
- Database connectivity is very well supported.
- The development environment supports easy testing and debugging.

7.2.2 System Architecture

Figure 43 shows the architecture of HB-CAS in terms of its modules and files. The diagram shows the data flow within the system but omits the options files that some modules possess.

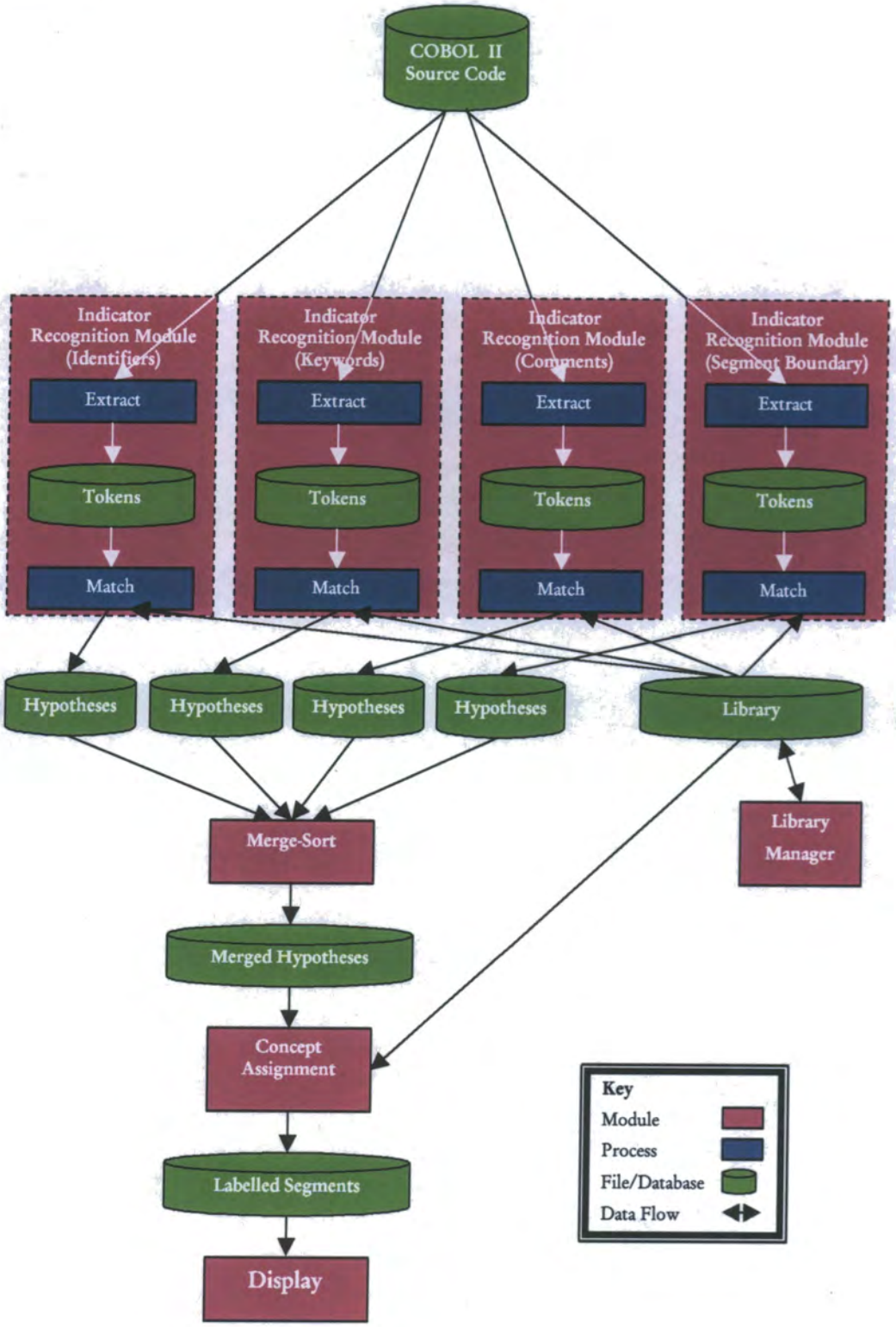


Figure 43: Architecture of HB-CAS, Showing the Data Flow between Modules and Files

The architecture reflects the design of the HB-CA method. The sort and indicator recognition modules encapsulate the hypothesis generation stage, and the concept assignment module performs segmentation and concept binding. The design of HB-CAS has been influenced by a number of considerations:

- **The need to control the system easily.** This has been met by using a control panel to monitor and manage system execution. The control panel validates files, sets library information, changes module options, and allows access to intermediate data files during execution. The control panel provides the user with a single interface to all parts of HB-CAS. It is intended for the expert user or system developer and would need modification if the product were to be used in other situations. It is unlikely that an average software maintainer would need or desire the level of information that can be gained from the control panel, but would be more interested in the results of the process. The control panel also permits easy expansion of the indicator recognition part of the system. Indicator recognition modules can be added and removed without the need to inform the control panel explicitly since it detects their presence dynamically. The control panel also enables each module to be executed individually and the combination of modules can be changed. This allows different sets of a module's options to be used without re-executing modules preceding the one being tested. Figure 44 shows the control panel.

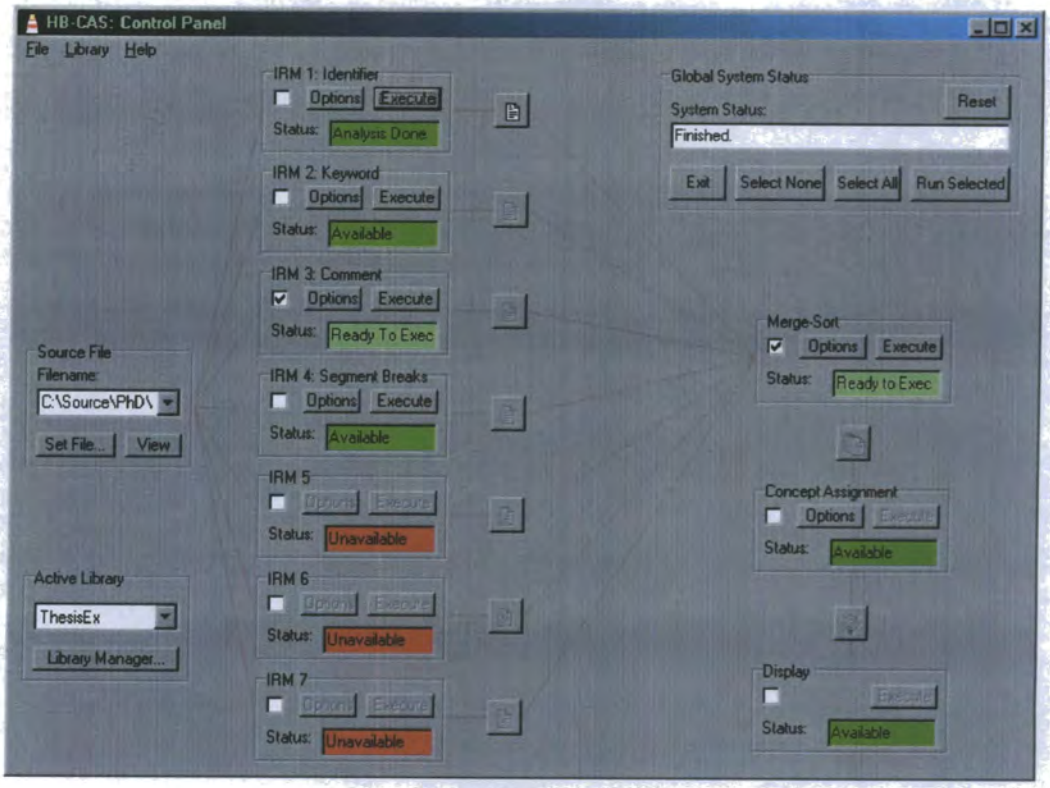


Figure 44: HB-CAS Control Panel

- **The need to permit swift modification of system modules.** As the techniques within HB-CA were improved, various parts of the system required modification at different times. This led to a system design composed of a large number of stand-alone programs linked by the control panel. Each program takes a number of command line options and in some cases reads an options file. Adopting this approach meant that when a module was changed, it was the only program requiring recompilation, the rest of the system remaining unaffected.

- **The need to access intermediate representations during development.** One of the consequences of employing a separate module approach is that files need to be used as an intermediate data structure. This provides the user with easy access to the data available between module executions. Using files can have an adverse effect on performance although this has not been particularly apparent during development and evaluation.

7.2.3 Library Structure and Management

The library is stored as a Paradox 7.0 relational database constructed using the Database Desktop utility of Delphi 4.0. The structure of the database reflects the library structure described in Chapter 3. This is depicted in Figure 45.

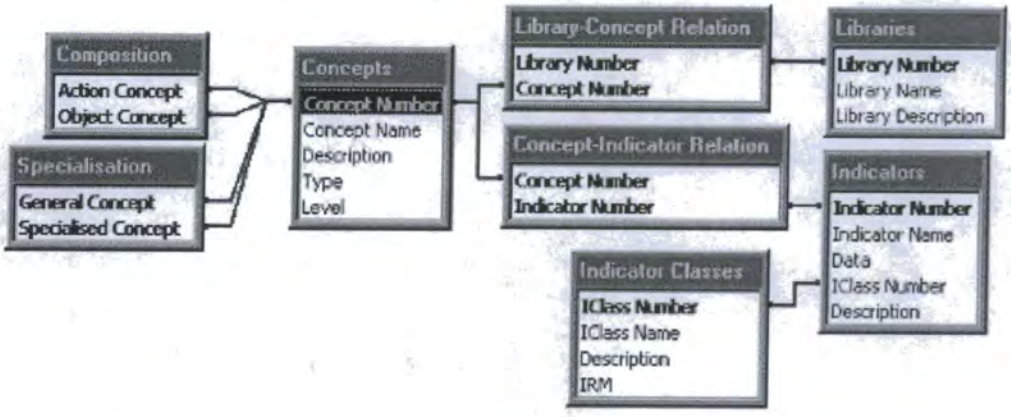


Figure 45: Library Structure Implemented in Relational Database

The database is managed by a small module called the Library Manager. This permits the addition, deletion, and modification of concepts, indicators, and the relationships between them. Although some validation is performed within the module, users often have access to the raw database tables underlying the front-end. If the system was released for large-scale use this would need to be rectified, however the current situation is acceptable for a research prototype. The Library Manager module is shown in Figure 46.

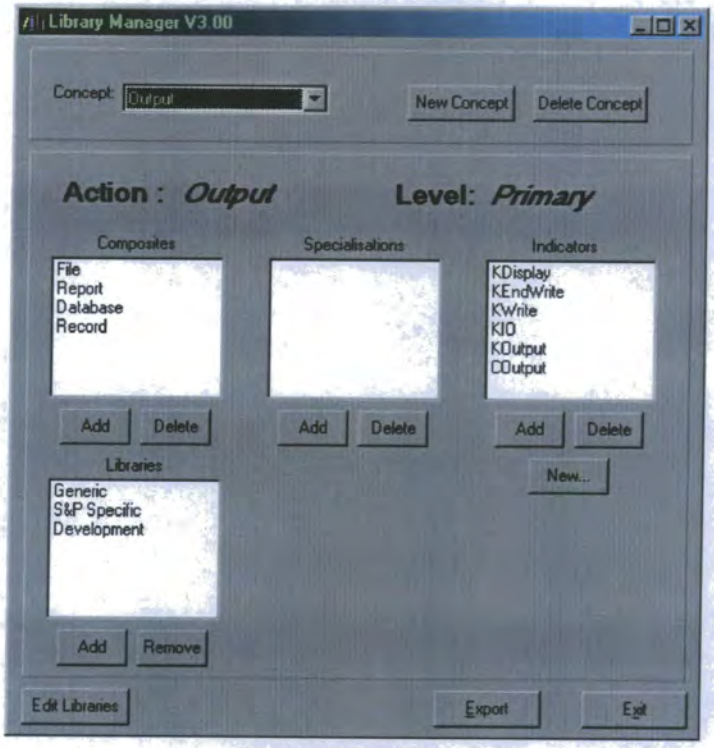


Figure 46: HB-CAS Library Manager

The Library Manager permits concepts to be assigned to more than one "library". The control panel requires one "library" to be selected as active for a particular execution. This allows a good degree of flexibility for the user since separate sets of library content for different applications can be developed from a common core of concepts.

7.2.4 File Formats

Due to the nature of the data passed between modules in the system, the file format requires a well-defined structure. This needs to be capable of storing multiple attributes about each data item, whether a hypothesis, indicator, or conclusion. The Microsoft Windows INI file lends itself well to this application. Data items are referenced by a string enclosed in square brackets, and a series of name and value pairs contain the item's attributes. An example of an INI file entry is shown in Figure 47.

```
[0]
Line=25
Pos=15
IClass=1
Token=C10-INITIALISE
```

Figure 47: Example of an INI File Entry

Although obsolete in Windows 95/98, having been replaced by the registry, the INI file is still supported in the Win32 API. Delphi provides a class wrapper for the INI file in its own libraries. This provides the programmer with pre-defined routines for storing and retrieving information in the files using random access. The advantage of this approach is that no parsing code needs to be written to read the contents of the files, and the structure is clear enough for the developer to read the contents without translation.

INI files are limited to 64Kb in size. Although most parts of HB-CAS do not create files that come near the limit, the combined data from indicator recognition was found to reach this point occasionally. Delphi's INI file implementation allows a memory-based version to be used without any size limit. This has the additional advantage of significantly increasing the performance of the module using it.

7.2.5 Indicator Recognition Modules

Each indicator recognition module extracts indicators in one of the classes discussed in Chapter 4. The modules operate in two stages: extraction, and matching. The extraction modules are written in C and are simple lexers. The basis for each lexer is a commercial lex/yacc package written for COBOL 85. This has been extended for IBM COBOL II. The lexers extract all procedure division tokens falling into the appropriate indicator class, with the exception of the comment recognition module. During the development of HB-CA, comments occurring before the procedure division were required and the module was designed to extract them. As the segmentation method shown in Chapter 5 ignores any hypotheses generated from this part of a COBOL II program, no modifications have been made to the module. These tokens can still cause hypothesis generation but the hypotheses are not considered in segmentation and concept binding. The result of extraction is matched against the database of indicators.

The matching stage is written in Delphi and each module uses a file of options to determine its behaviour. Case sensitivity in search is available within the Delphi database components so no custom implementation is required. Sub-string matching requires a bi-directional comparison of the database string with the token extracted from the source code. Synonym matching was implemented using Automation links to Microsoft Word. Word exports functions in its type library to access the thesaurus and these were used to generate synonyms.

The separate outputs of the indicator recognition modules are combined to produce a file of all the hypotheses made. This file is sorted in order of indicator occurrence.

7.2.6 Concept Assignment Module

The concept assignment module implements the segmentation and concept binding algorithms described in Chapters 5 and 6. The major point of interest in this module is the implementation of the self-organising map. Kohonen's research group provides a self-organising map implementation for MS-DOS called SOM_PAK. It is available on the web [SOMP00] and provides a suite of programs for creating, training, and interpreting SOMs. Rather than implement a native Delphi version of the SOM algorithm, it was decided to use the SOM_PAK and harness it to the Delphi program through an MS-DOS batch file interface. This is less efficient than a native version but has the advantage of using a proven implementation. SOMs can be initialised in a number of ways, the most efficient being based on eigenvectors [KOHO00]. The method used in HB-CAS is random initialisation, as described in the SOM_PAK documentation [KOHO96].

It is important to establish the reliability of a third-party implementation. Establishing confidence in the SOM_PAK was achieved by experimenting with simple maps to successfully produce predicted results. In addition, the SOM_PAK has been used in a variety of research projects with no reported problems (see [LAGU96], [HAME96], [VESA97], [DESJ00]).

7.2.7 Display

The display module provides the user with a hypertext-style view of the source code and concepts. Concepts are coloured and these correspond to segments of the same colour in the source code. Clicking on a concept name scrolls the code to display its implementation. The display system only provides browsing of the source code, as it is expected that the search methods contained in the system would be integrated into a full development environment for real-world release. Figure 48 shows the display module.

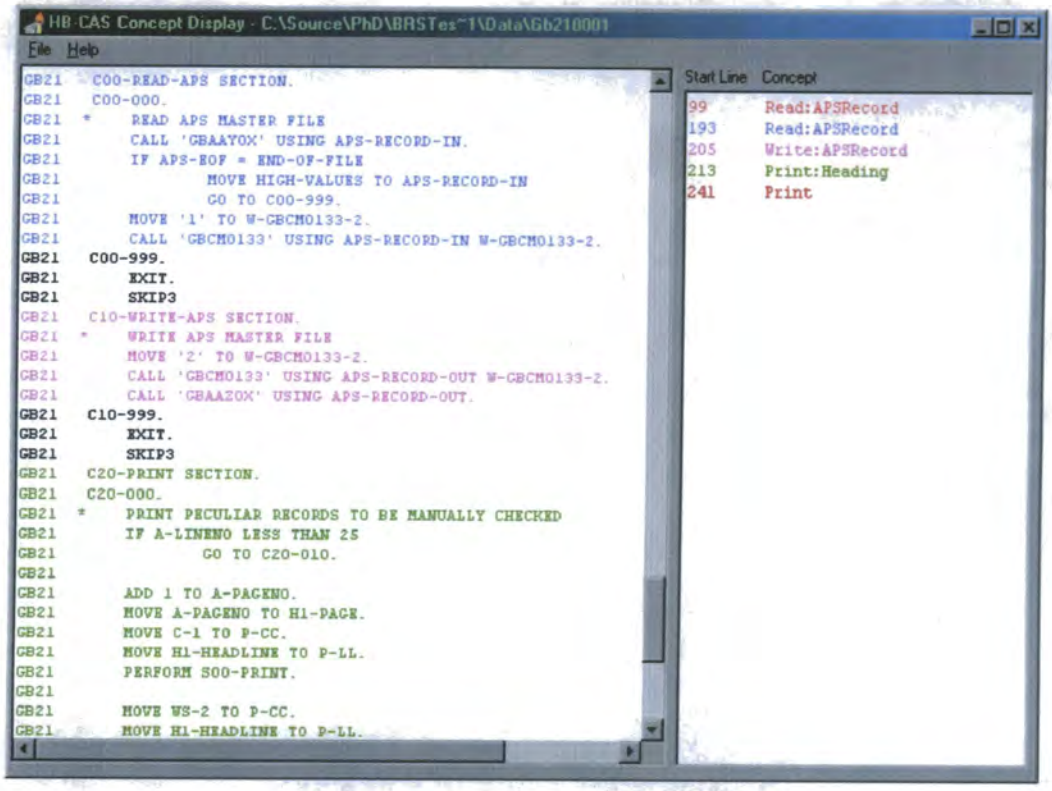


Figure 48: HB-CAS Display Module

7.3 Test Suite

A test suite was constructed to interface with HB-CAS. It allows a wide range of parameters to be tested in a controlled manner and was employed during the development of HB-CA to examine the relative performance of the various approaches.

7.3.1 Principles of Test Suite

The test suite allowed the concept assignment system to be executed on large numbers of source files with varying parameters. The results of each execution were recorded and compared with predefined "correct" answers. This allowed automatic measurement of the system's success at concept assignment. A log file stored the input and output of every stage of the recognition process for later analysis. The size of this file (typically between 22Mb and 50Mb) required the development of several *perl* scripts to create summaries of the information contained within it.

To ensure fair testing, the suite controls HB-CAS using the control panel "execute" method. Its only direct intervention in the process is to set module options in the relevant initialisation files. Once this has been completed, control is passed to the control panel as if a person was using the system.

Creating the "correct" answers required the use of a mark-up tool to designate parts of the source code as indicators and concepts. The marked-up code was stored in files and automatically compared to the output of the concept assignment system using various criteria for correctness. Performance was measured in terms of information retrieval, using *precision* (number correct/number found) and *recall* (number found/number potential). The best set of indicator recognition options was determined by comparing the precision and recall values for each combination. These investigations also guided the development of the concept assignment methods.

As the concept assignment methods became more sophisticated, the marked-up representation of the source code became incompatible with the output of HB-CAS. Since later versions of HB-CAS had fewer parameters to control the concept assignment process, the effort required to upgrade the test suite was not deemed worthwhile. Consequently, the evaluation of HB-CA described in the next two chapters is based on the manual application of correctness measures.

7.3.2 Usage

Although the test suite was originally intended for use as an automated test and evaluation tool, it could be adapted easily to provide batch-processing facilities. This would allow large amounts of source code to be analysed in one step, either for future software comprehension use (storing the results in a repository), or for more specialised applications such as searching for instances of a particular concept in a large body of source code. This could fulfil the functions of the wrapper, discussed in section 9.2.1, for ripple analysis, module selection, and code reuse.

7.4 Evaluation of Implementation

This section discusses some of the issues arising from the implementation of HB-CA in HB-CAS. The method itself is evaluated in the next two chapters.

7.4.1 Design Evaluation

7.4.1.1 Separate Program Approach

The basic design has proved effective and the architecture has not been changed for any version of HB-CA. Using separate programs to implement each part of the system made modification and testing easy. The control panel's ability to scan the directory structure for additional indicator recognition modules also helped to expand the system with minimal effort.

There were some disadvantages, in particular, the problem of synchronisation between the control panel and the other programs. This was handled using the presence of a file to act as a "process complete" flag. This somewhat inelegant solution could be replaced with the Win32 process control API, but the effort required to understand and employ these functions was judged greater than the potential benefit. The API method offers better performance and greater elegance, but the file-based method works satisfactorily.

The individual modules run in "batch" mode rather than in the traditional interactive manner of Windows applications, ensuring that the control panel does not require additional user input once the run button is selected. Adopting this approach places responsibility on the programmer to ensure that the application

window is updated and operating system messages are processed. This problem would be overcome by implementing HB-CAS as a single program.

7.4.1.2 Third-Party SOM Implementation

There were good reasons for using the SOM implementation provided by Kohonen's group, the most important being that the code can be trusted as correct (see section 7.2.6 for research citing use of the SOM_PAK). In addition, a substantial amount of time was saved by not re-implementing the algorithms.

When SOM-based methods were first employed, the techniques of file-based synchronisation were well understood and a Delphi library was built to interface with the SOM_PAK. Using a separate library to abstract SOM functionality provides an easy way to substitute a native implementation should the need arise.

The main disadvantage of using the SOM_PAK is poor performance. The programs compile to a DOS executable and consequently require a command shell to be launched before execution. In addition, different programs within the package handle the stages of initialisation, training, and interpretation separately. This leads to a new shell being launched for each. Despite this high run-time overhead, performance on real data is acceptable, although a native 32-bit implementation would almost certainly show significant performance gains.

7.4.1.3 Third-Party Synonym Lists

Synonym-based indicator matching is not used in the examples in this thesis as it significantly degraded indicator recognition performance and computational cost. The idea of using synonyms to give flexibility to indicator recognition is considered sound, although better methods are required to perform the matching process.

Microsoft Word was chosen to provide the synonym list since it has wide availability on the Windows platform and a standard library with which other programs may access its functionality. Word list quality was not considered in this research but if synonym matching is desired then list quality should be addressed.

7.4.2 Code Evaluation

7.4.2.1 System Characteristics

Table 9 presents some general characteristics of the HB-CAS implementation. Compiled sizes are given to the nearest kilobyte for the debug version of modules. This version has been used throughout the implementation and evaluation.

Program	Source Length (lines of code)	Compiled Size (Kb)	Source Language
Identifier Extraction	1687	67	C
Identifier Matching	301	507	Delphi 4
Keyword Extraction	1694	68	C
Keyword Matching	176	513	Delphi 4
Comment Extraction	136	25	C
Comment Matching	265	529	Delphi 4
Segment Boundary Extraction	1718	68	C
Segment Boundary Matching	167	517	Delphi 4
Control Panel	1289	1205	Delphi 4
Library Manager	776	689	Delphi 4
Merge-Sort	230	313	Delphi 4
Concept Assignment	1633	562	Delphi 4
Display	410	367	Delphi 4
Total	10482	5430	

Table 9: Characteristics of HB-CAS Programs

7.4.2.2 Programming Environment and Language

Delphi has proved to be an excellent language and environment within which to work. Its rapid prototyping capabilities removed much of the effort of user interface design and management, and a good debug environment helped with testing. An almost perfect balance is struck between abstraction from the Windows API and providing flexibility in library routines. In addition, it has excellent database connectivity that made accessing the library extremely easy.

7.4.3 Test and Validation

Each program was tested individually before being included in the system. The separate program approach and high visibility of input and output data meant that very few problems were found during integration. Individual programs were mostly checked by hand to ensure that the output generated was as expected, e.g. the extraction and match program results were compared to a manually performed analysis.

The concept assignment module was more complex and required the use of Delphi's debugging tools. These allowed the internal state of various data structures to be displayed at appropriate points during the execution of the module. Single-step tracing of the routines was used to ensure correct implementation of the algorithms.

Due to its nature, the specific behaviour of the SOM cannot be accurately predicted, but experiments during the development of HB-CA gave an indication of typical results. These were used, in conjunction with test data, to verify that the SOM was working as expected.

Despite thorough testing before evaluation, the investigations undertaken for Chapter 8 highlighted a few remaining bugs when the more complex library content (shown in the Appendix) was used. These were rectified without significant effort, and the affected investigations repeated with negligible difference in their results.

7.5 Summary

This chapter has presented the HB-CAS implementation of the HB-CA method. Various technical issues relating to the system's design have been discussed and its automated test suite described. The implementation has been evaluated with respect to major design and code characteristics.

Chapter 8 presents the first part of an extensive evaluation of the HB-CA method. This examines many characteristics of HB-CA, beginning with its scalability.

Chapter 8
Evaluation I: HB-CA Characteristics

8.1 Introduction

Chapter 7 described an implementation of the HB-CA method called HB-CAS and discussed its architecture and design rationale.

Having shown the operation of each stage of HB-CA in Chapters 4, 5, and 6, this chapter presents the first part of an extensive evaluation of the method, relating to characteristics of HB-CA itself.

The evaluation begins with one of the most important properties of HB-CA: scalability. HB-CA is intended to work with real-world code and hence it is important that it operates accurately on any length of program. The chapter then discusses issues relating to segmentation, concept binding, and the library. Finally, some general characteristics of HB-CA are examined: computational and spatial cost, expandability, representational power, domain independence, and achievement of cognitive requirements.

The results of a number of practical investigations are reported, each introduced by a table summarising its parameters and data. Investigations were carried out using HB-CAS and a number of real-world COBOL II programs. Program sets, results, and other parameters for all the investigations can be found in the Appendix. All program lengths are quoted in lines including white space and comments, since these can contain valid indicators. Although HB-CA is designed to work solely on the procedure division of COBOL II programs, it is not reasonable to expect a maintainer to remove the data division before commencing analysis. Consequently, program lengths include the data division, and all investigations use complete programs.

8.2 Scalability

Accurate concept assignment is important since mistakes could confuse the software maintainer, thus increasing, rather than decreasing, the cost of software comprehension.

HB-CA should maintain its accuracy regardless of the length of program to which it is applied. In principle, if HB-CA can be accurate on a single segment, there is no reason why it should be inaccurate when there are several segments, as each is analysed separately.

Concept assignment is regarded as *accurate* if a segment implements the concept specified. Figure 49 shows an example.

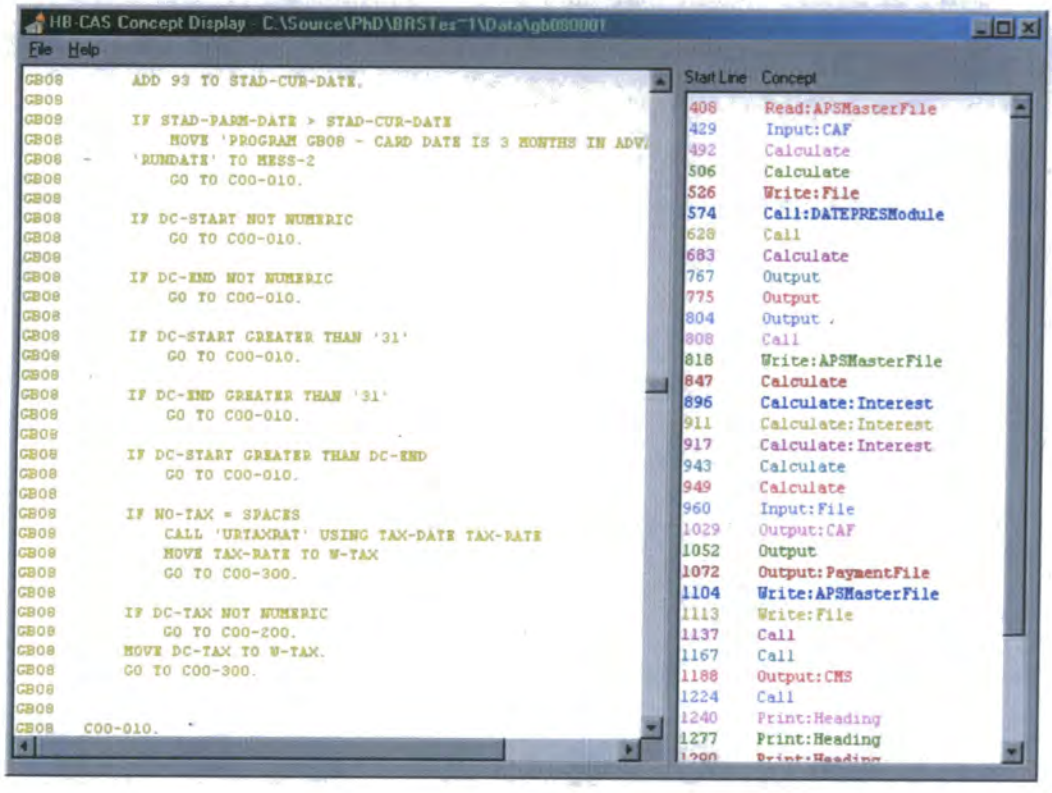


Figure 49: Example of an Accurate Segment

In this case, the segment is assigned the concept Call starting at line 628 (shown in beige). This is classed as accurate because a call is made from this section of code although the remainder of the segment is concerned with other processing.

Concept assignment is regarded as *strictly accurate* if the concept is dominant in the segment (i.e. the segment is mostly concerned with implementing the concept specified). Figure 49 is not strictly accurate as it is concerned with program control rather than with calling. An example of strict accuracy is shown in Figure 50.

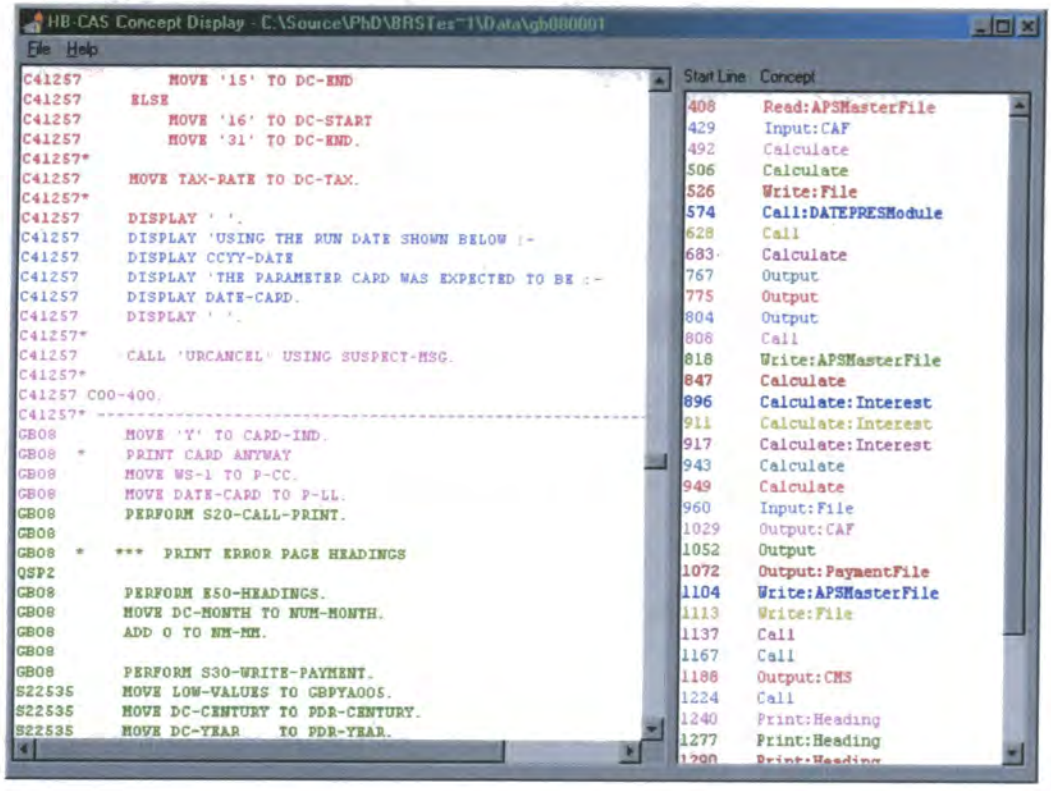


Figure 50: Example of a Strictly Accurate Segment

It can be seen clearly that the blue segment, which is assigned the Output concept starting at line 804, is implementing only this concept. It is therefore dominant in the segment and strictly accurate.

Recall that *rec_thresh* is used to ensure a certain level of evidence in each segment (see section 5.3.2.1), and to determine the lowest score with which a concept may be bound during concept binding (see section 6.3.2.4). The minimum vector density required for a cluster in the SOM to be classed as valid (see section 5.3.2.3) is defined by *min_vd*. This parameter is used also to determine the number of potential clusters in a segment during clustering pre-processing (see section 5.3.2.1). *Forced_specialisation* determines whether post-disambiguation processing (see section

6.3.2.4) should attempt to find the most specific version of the winning concept for which there is evidence.

To verify the scalability of HB-CA, an investigation was undertaken using HB-CAS. The parameters are shown in Table 10.

Program Set	Set A
Library Content	Appendix, Section A.2
rec_thresh	1
min_vd	3
forced_specialisation	True/False
Results	Appendix, Section A.3.1

Table 10: Parameters for Investigation of Scalability

The results of the investigation are shown in Figure 51 and Figure 52.

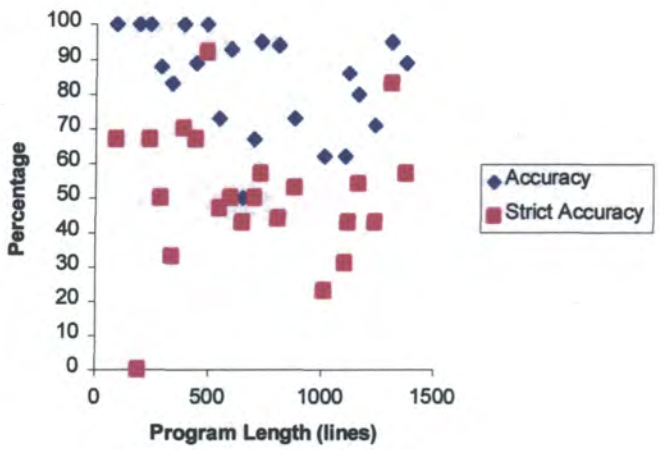


Figure 51: Graph to show the relationship between the Accuracy of Concept Assignment and Program Length (*forced_specialisation* = True)

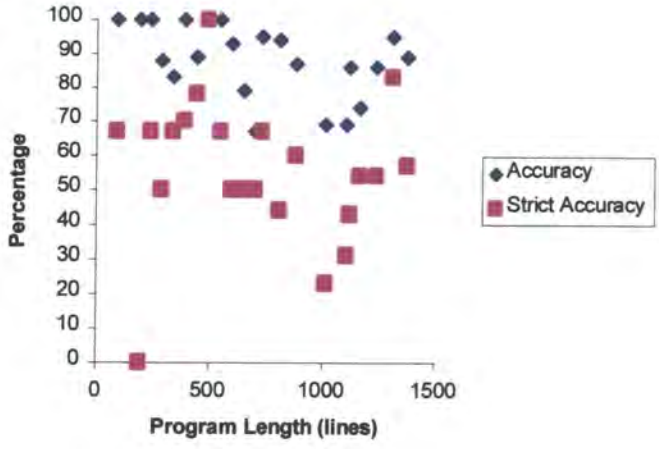


Figure 52: Graph to show the relationship between the Accuracy of Concept Assignment and Program Length (*forced_specialisation* = False)

Each point on the graph represents the accuracy (or strict accuracy) for a particular length of program. The set of programs used is shown in section A.9.1. Both accuracy and strict accuracy are plotted on the same set of axes for easy comparison.

Although both graphs show a wide variation in their results, the general trend does not confirm the theoretical claim that accuracy should remain the same regardless of program length. Forcing specialisation produces slightly less accurate results although those concepts that are correct should provide more information to the user. Accuracy drops significantly at a program length of about 1000 lines, strict accuracy following a similar trend.

8.2.1 Investigation of Scalability Problems

Recall that the segmentation stage of HB-CA creates segments initially using segment boundary hypotheses. Each segment is then analysed to determine the potential for forming clusters of action-concept hypotheses within it. If such potential exists, a SOM is used to cluster similar hypotheses. Those clusters that have sufficient vector density are termed valid, and those that do not are termed invalid. Invalid clusters are equally divided and merged with their nearest valid neighbours. Each cluster is then created as a segment in its own right and the object concepts that fall within and around its boundaries are included.

Accurate concept binding relies on a good quality segment, i.e. a set of hypotheses that clearly indicate one concept. It follows that the lower a segment's quality, the less likely the concept binding method is to accurately assign a concept. The following hypothesis is made to explain the drop in accuracy with larger programs:

Hypothesis 1: Segmenting larger programs requires greater use of SOMs, which reduces the accuracy of concept assignment.

The first question to be addressed is whether larger programs use more SOMs. To test this, comparisons are made between the length of programs and the number of SOMs used to analyse them. Results are taken from the investigation summarised in Table 10. Figure 53 shows that SOM usage increases when larger programs are analysed.

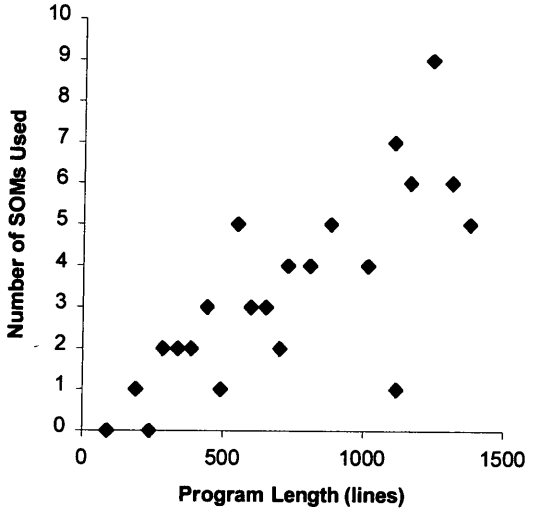


Figure 53: Graph to show the relationship between the Number of SOMs Used and Program Length

Since larger programs do require greater use of SOMs, it is likely that the latter is the cause of lower accuracy. Further confirmation is gained by comparing SOM usage and accuracy directly, as shown in Figure 54.

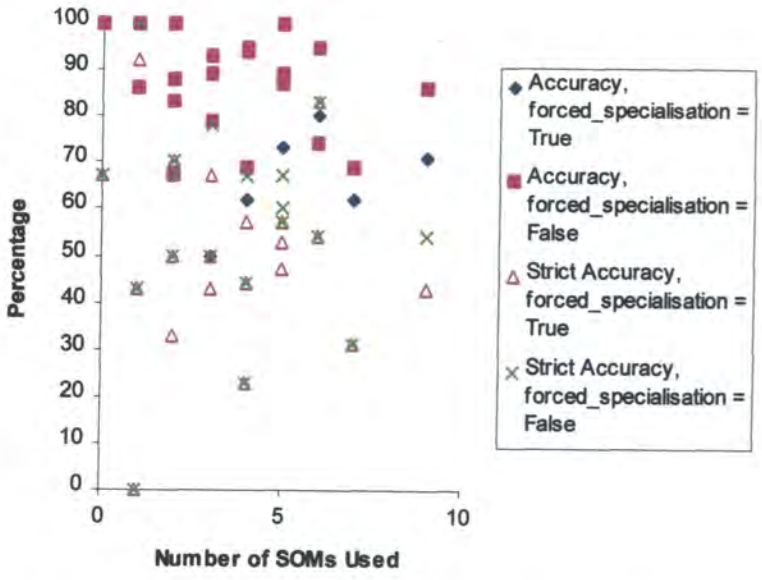


Figure 54: Graph to show the relationship between the Accuracy of Concept Assignment and Number of SOMs Used

The data indicates that the greater SOM usage arising from analysing larger programs correlates with a reduction in the accuracy of concept assignment. Hypothesis 1 would appear to be confirmed.

Recall that low concept assignment accuracy can be caused by poor quality segments. A further hypothesis is made to explain why greater SOM usage causes lower accuracy:

Hypothesis 2: SOM usage causes lower quality segments.

If the hypothesis were correct, it would explain the fall in accuracy with greater SOM usage. Hypothesis 2 is investigated in the next section.

8.2.1.1 SOM-Related Segmentation Problems

This investigation begins by discussing the relationship between accuracy and segment size.

Observation of the code segments that were assigned concepts indicated that the most accurate assignments were made when the segment size was small. Smaller

segments tended to occur in smaller programs. Since lower accuracy is linked to large programs, it could be that SOM usage causes a rise in segment size. This could explain the fall in accuracy when more SOMs are used. Smaller segments are likely to contain fewer hypotheses and consequently there is less potential for confusion. The segment quality therefore is higher and accurate concept binding more likely to result.

In an attempt to verify these observations, and perhaps find the optimal size of a segment, several programs were examined for the size of their segments and the accuracy of the concepts bound to them. The investigation parameters are shown in Table 11 and the results are shown in Figure 55.

Program Set	Set B
Library Content	Appendix, Section A.2
rec_thresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Section A.3.2, A.3.3

Table 11: Parameters for Investigation of Segment Size and Accuracy

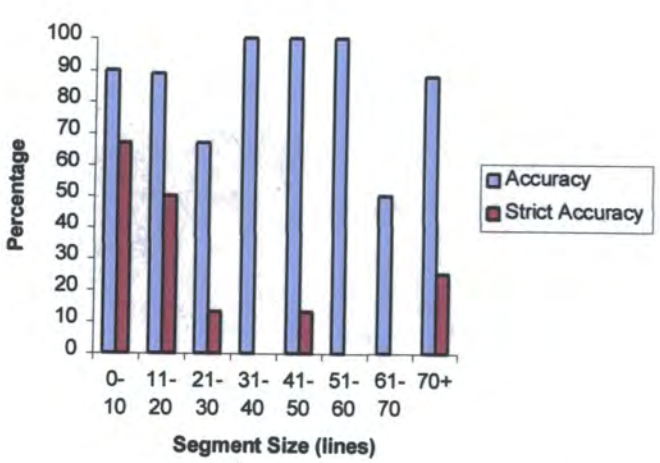


Figure 55: Chart to show the Accuracy of Concept Assignment for Various Segment Sizes

Accuracy peaks with segment sizes of 31-60 lines. The presence of similar accuracy levels elsewhere on the chart means that drawing a firm conclusion that this is the ideal segment size would be unwise. Strict accuracy is clearer, with the best results definitely coming from the smallest segment sizes. This confirms the observations reported at the start of this section.

Figure 56 shows further results from the investigation in an attempt to determine a relationship between SOM usage and mean segment size. The numeric results are shown in section A.3.3.

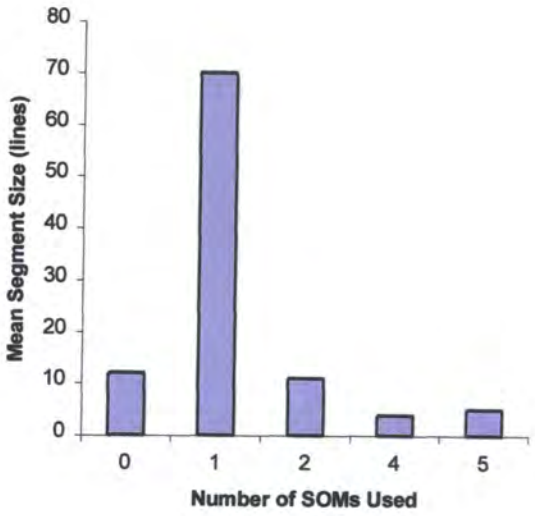


Figure 56: Chart to show the Mean Segment Size for Various Numbers of SOMs Used

It is clear from Figure 56 that there is no correlation between greater SOM usage and greater mean segment size. Consequently, the explanation for the fall in accuracy with larger programs must be attributed primarily to a reason other than larger segment sizes.

Two further explanations for the link between greater SOM usage and lower quality segments are:

1) The SOM is associating concepts that should not be clustered.
2) The algorithms that reallocate action-concept hypotheses from invalid clusters (see section 5.3.2.4) are introducing enough unrelated concepts to valid clusters to cause poor segment quality.

The most likely explanation can be determined by studying the balance between valid and invalid clusters at varying accuracies. If a low proportion of invalid clusters correlates with low accuracy, this would suggest that the SOM is causing the problem because the reallocation algorithms are not being used to a great extent. If there were a link between a high proportion of invalid clusters and low accuracy, this would indicate that the reallocation algorithms are at fault because they are being used often.

An investigation was undertaken on various programs that require SOM analysis. Sections that were subdivided by a SOM were examined to determine the number of valid and invalid clusters produced, and the accuracy of concept assignment for each resulting segment. The reintegration of object concepts is of less concern since they can only confirm and complete conclusions, not generate them initially. Confusion in object-concept hypotheses has less impact on the correctness of the result owing to the disambiguation rules. The investigation parameters are shown in Table 12.

Program Set	Set C
Library Content	Appendix, Section A.2
rec_thresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Section A.3.4

Table 12: Parameters for Investigation of Accuracy and Invalid Cluster Proportions

Figure 57 indicates that higher proportions of invalid clusters lead to lower strict accuracy, although this is not reflected to the same extent in non-strict accuracy.

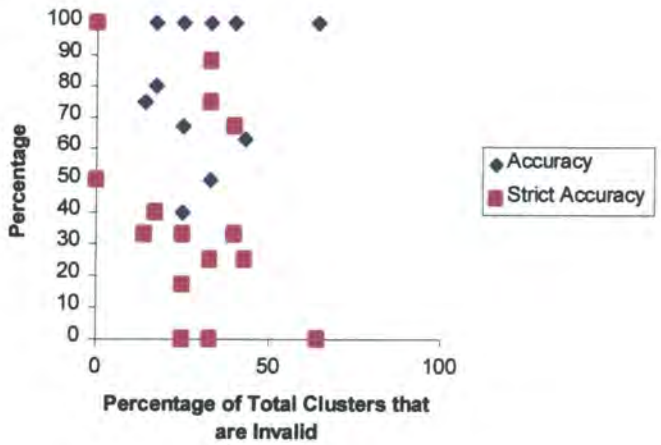


Figure 57: Graph to show the relationship between the Accuracy of Concept Assignment and the Proportion of Invalid Clusters

Since non-strict accuracy relies only on the presence of a concept within a segment, good results can be achieved with poorer segmentation. The requirements of strict accuracy mean that "loose" segmentation (where a large part of the segment is irrelevant to the concept) is more evident in the results. The conclusion that can be drawn is that the problems lie in the reallocation algorithms. This is not surprising since the "equal-division" method of assigning invalid clusters and hypotheses to their surrounding valid clusters is naïve. It causes "loose" segmentation by including hypotheses in segments to which they may have no conceptual affiliation, and adding entire invalid clusters to their neighbours without considering the content of either. When considering the problems the latter may cause, it is worth recalling that the SOM has associated the hypotheses in an invalid cluster, and consequently the neighbouring valid cluster gains a conceptually coherent group of hypotheses. Concept binding then could be hampered by both the general "noise" of unrelated individual hypotheses, or worse, it could be led in a completely different direction by conceptually coherent, but unrelated, groups of hypotheses.

8.2.1.2 Possible Solutions

The reallocation algorithms would benefit from further research. One approach might be to use conceptual information from the hypotheses of invalid clusters, to bind them to conceptually similar neighbours. This might require some preliminary concept binding. Alternatively, the principle of preserving all of the original hypotheses could be rejected and invalid clusters ignored. Another idea might be to limit the number of hypotheses that can be added to a valid cluster, or limit the cluster size itself.

Another approach to improving the quality of segmentation might be to change the controlling parameters, *rec_thresh* and *min_vd*, which for the investigations performed in this chapter are set to 1 and 3 respectively. Increasing *rec_thresh* would cause a reduction in the number of initial segments and hence concept assignments made (since more evidence would be required). Those segments that pass the threshold would be larger, having a reasonable amount of evidence. Smaller values of *rec_thresh* would allow more initial segments to be considered and increase the number of concept assignments. Given that smaller segments have been observed to produce more accurate concept assignment, smaller values of *rec_thresh* should produce more accurate results overall. The disadvantage of having smaller segments is that each hypothesis carries more weight (by representing a larger proportion of the body of evidence) than in larger segments. Consequently, a misleading indicator can cause greater problems. Individual hypotheses in larger segments have less influence on the overall concept assignment, so increasing *rec_thresh* may ensure that a reasonable body of evidence is considered, rather than just a few hypotheses.

Increasing *min_vd* would increase the number of invalid clusters by forcing valid clusters to contain more evidence. This could cause poorer segmentation for the reasons discussed in section 8.2.1.1. Decreasing *min_vd* may improve the quality of segmentation, but the resulting segments could be so small (since only one or two hypotheses for a concept would be required) that concept assignment would become pointless. There would no longer be a significant body of evidence to consider (see the discussion of *rec_thresh* above). A balance must be struck when setting the parameters, to make best use of the library on the source code being studied.

8.2.1.3 Summary

There is a clear link between SOM usage and poor segment quality. Hypothesis 2 has thus been confirmed. Poor quality segments result from the application of the naïve reallocation algorithms. Greater SOM usage results in a greater chance of these algorithms being employed. This explains the fall in concept assignment accuracy when more SOMs are used.

It should be noted that in some cases, SOM-based segmentation is very successful and further discussion is presented in section 8.3

8.2.2 Average Performance

The overall performance of HB-CA is promising, achieving high mean and median accuracies as shown in Table 13.

	forced_specialisation = True	*forced_specialisation* = False
Mean Accuracy	84%, σ = 14	88%, σ = 11
Mean Strict Accuracy	56%, σ = 19	56%, σ = 21
Median Accuracy	89%	89%
Median Strict Accuracy	50%	56%

Table 13: Average Accuracy Values for HB-CA

It is interesting to note that using general versions of concepts (when *forced_specialisation* is False) does not increase the accuracy significantly. This suggests that the concept binding algorithm is capable of successfully differentiating specialised concepts, or that the library has little specialisation.

8.2.3 Summary

Despite theoretical claims that accuracy should not decrease with longer programs, investigations indicate that such programs cause a wider variation in accuracy and a general drop in concept assignment performance. This is attributed to the greater use of SOMs when analysing larger programs, and the poorer quality of segmentation that can result. Small segment sizes appear to provide the best

recognition performance when strict accuracy is considered, although non-strict accuracy does not indicate this to the same extent.

Investigation of the cause of SOM-related segmentation problems revealed that the hypothesis reallocation algorithms are largely to blame for poor performance. This is not surprising given their naïve nature and several strategies have been identified to address the problem.

8.3 Segmentation

In view of some of the issues raised about HB-CA's segmentation in the previous section, a discussion of the abilities of the segmentation method is now presented.

HB-CA is designed to operate on real-world code and consequently cannot rely on being applied to well-structured programs. When poorly structured code is presented, SOMs are used to create segments based on conceptual association rather than syntactic boundaries. Section 8.2 discussed some of the problems that arise when SOMs are employed in this role. These appear to be linked mostly to the algorithms that analyse the results produced by the maps.

A small investigation of the SOM's ability to replicate syntax-based segmentation has been performed. Pairs of segment boundary hypotheses were removed successively from the hypothesis list. In the programs tested, the SOM failed to preserve the syntactic clustering exactly, although the resulting concept assignment was still correct. The cross-subroutine segmentation that can occur in these cases was the motivation for the use of segment boundary hypotheses.

In some cases SOM-based segmentation can be extremely successful, an example being shown in the second and third concepts of Figure 58 (specialisation was not forced).

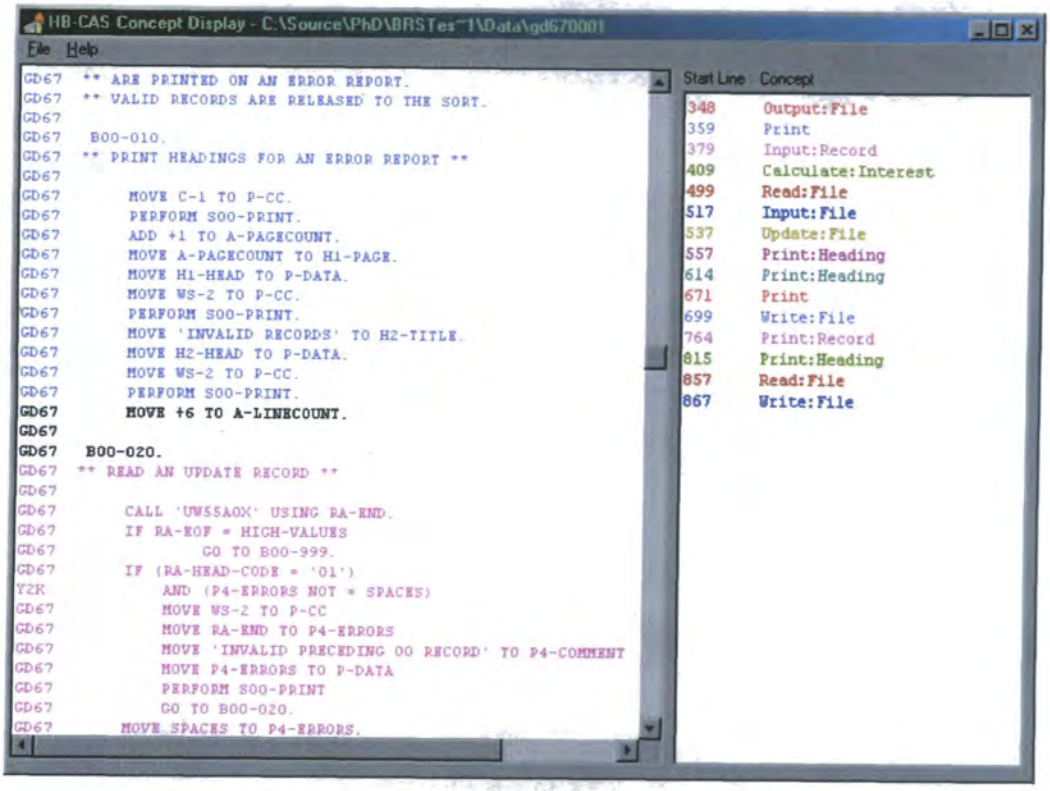


Figure 58: Screenshot Showing Successful SOM-Based Segmentation

In this example the segments are reasonably well focussed around the concepts they represent and there is good separation between them. This is how the operation of SOM-based segmentation was originally envisaged. On some occasions the SOM produces strange results. A common fault is that adjacent segments are created within a section, but all performing the same task (an example is shown in Figure 59).

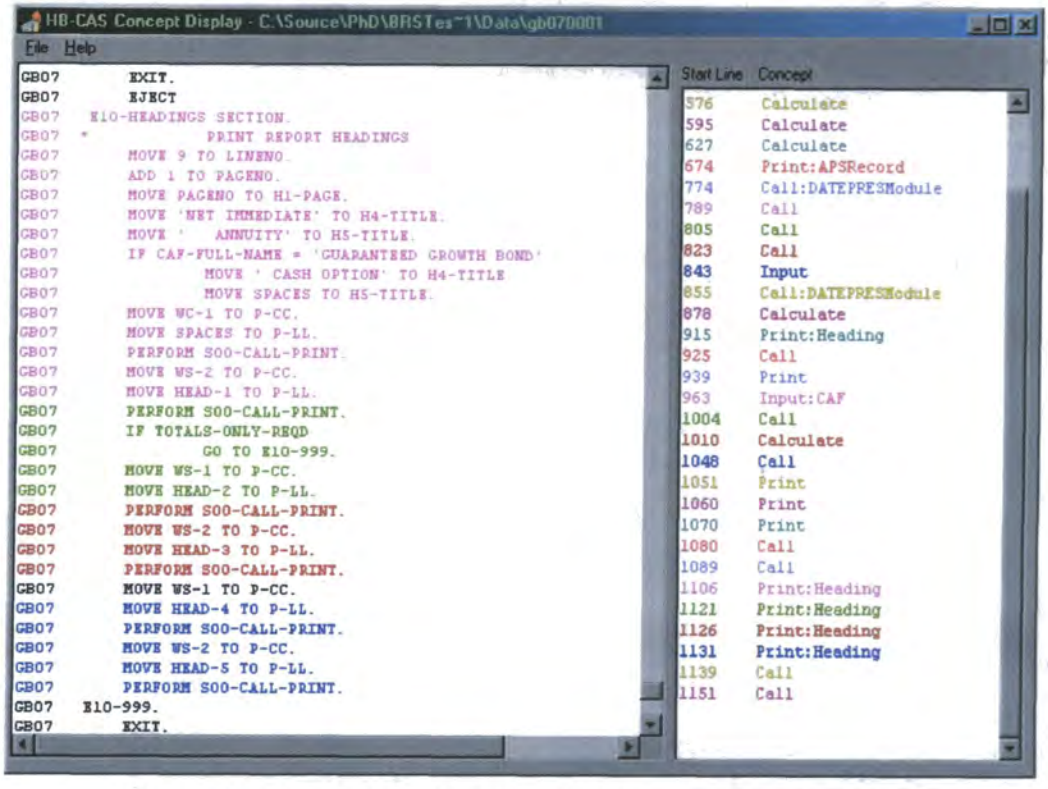


Figure 59: Screenshot Showing Unnecessary Segmentation

In this case, all of the concepts are correct but three are adjacent and identical. As they all perform the same task, there seems little point in regarding them as separate segments. The causes of this problem became apparent during the development of HB-CA and are related to the nature of the SOM itself. One explanation is that rather than associating all of the hypotheses to the same output node and quantizing the few unrelated hypotheses, the SOM learns to differentiate between regions of conceptual similarity and regions of conceptual difference. Consider a list of concepts:

Print, Print, Print, Read, Read, Read, Read, Print, Print, Print

When analysed by a 5x1 SOM, the result should be three triggered output nodes, each containing three or four of the same concept as appropriate. In practice, it has been observed that the SOM may group the concepts in pairs thus:

Print, Print Print, Read Read, Read Read, Print Print, Print

The SOM appears to learn both similarity and difference. This is essentially a problem of over-representation in the output space but since the method for calculating the size of the output space is well justified theoretically, there is little incentive for modifying it. An alternative explanation is that the initial values in the map cause concepts that should be associated to be pulled away from their correct cluster. The map training parameters should prevent this from happening in most cases.

A simple solution to the problem of unnecessary segmentation would be to add a rule to gather adjacent, identical concepts, and make one assignment for the collection.

In summary, SOM-based segmentation is a successful method for handling monolithic code and large subroutines, although it can over-segment on occasion.

8.4 Concept Binding

Section 8.2 raised the problem of poor concept assignment caused by hypotheses in a segment that are not related to the correct concept. This section examines the issue in more depth, evaluating the effectiveness of the disambiguation rules in allowing graceful degradation of the system's performance. The rules were presented in section 6.3.2.3.

Recall that concept binding operates by creating conclusions for every hypothesised action concept in a segment, and every possible composite containing that action concept. A composite is a concept made up of an action and an object. Object concepts are then scored to complete and strengthen conclusions. The highest scoring concept is designated the winner but if there are several with the same high score then disambiguation rules are applied to determine which should be ultimately successful.

Hypotheses that are unrelated to the correct concept in a segment can be viewed as falling on a scale from random "noise" to a coherent "interfering signal". If there is no conceptual correlation between these unrelated hypotheses, they simply create "noise" and are eliminated by the scoring algorithms. As the unrelated hypotheses

move along the scale to form a coherent "interfering signal", HB-CA must apply more rules to retain the correct concept. Eventually, the evidence for the interfering concept may outweigh that of the original and so assignment will be made to the new concept. This is not intrinsically bad since HB-CA was created to use the weight of available evidence to make concept assignments.

An example program was used to investigate the method's performance in this area, with the library content presented in the Appendix. *Min_vd* was set high to prevent a SOM being used. One routine with an obvious interpretation was chosen within the program, and varying types of unrelated indicator added.

The original routine is shown in Figure 60. The concept assigned by HB-CA (using *forced_specialisation* = True) was Read:PaymentFile. This is clearly correct.

```
GD25   S10-READ-PAYMENT SECTION.
GD25  *    READ THE PAYMENT FILE
GD25   S10-000.
GD25       CALL 'GBFDAMA0' USING PAYMENT-FILE.
GD25   S10-999.
GD25       EXIT.
```

Figure 60: Original Routine

The original routine contains only one unrelated indicator, CALL, and the scoring algorithms ignore this. FILE creates hypotheses for a number of different types of file in addition to the correct one, resulting in the bulk of evidence pointing to Read:File. When specialisation is forced, the hypotheses produced by PAYMENT indicate the result to be Read:PaymentFile. This can be seen in the extract from the HB-CAS log shown in Figure 61.

```
CB: Printing final conclusion list for this segment:
CB: Read  2
CB: Read:File  2:7
CB: Read:CAF  2:1
CB: Read:PaymentFile  2:4
CB: Call  1
CB: *DA* Finding highest scoring conclusions. (DAR 1)
CB: 1 high scoring conclusions.
CB: High scoring conclusion is Read:File, score 9
CB: *DA* Removing specialisations. (DAR 2)
CB: Found clear winner.
Winning conclusion for this segment currently Read:File
CB: ** Post-Disambiguation Processing **
CB: *PDAP* Specialisation Required.
CB: Found specialisation
CB: Found specialisation
Winning conclusion for this segment currently Read:PaymentFile
CB: *PDAP* Checking Thresholds.
Storing current winning conclusion
```

Figure 61: Extract from HB-CAS Log

The level of random "noise" in the original routine is moderately low. To test the ability of HB-CA to cope with greater "noise", the routine was modified to have more "noisy" indicators than indicators for Read:PaymentFile. This is shown in Figure 62.

```
GD25   S10-READ-PAYMENT SECTION.
GD25  *    READ THE PAYMENT FILE
NOISE *    PRINT A REPORT
NOISE *    UPDATE A DATABASE
GD25   S10-000.
GD25       CALL 'GBFDAMA0' USING PAYMENT-FILE.
NOISE      CALL 'PRINT' USING P-APS.
NOISE      MOVE SPACES TO DB-PARMS.
NOISE *    OUTPUT COMPLETE
GD25   S10-999.
GD25       EXIT.
```

Figure 62: Routine Modified with Random "Noise"

Despite the fact that there are only 7 indicators related to Read:PaymentFile and at least 9 unrelated, the system still makes the correct concept assignment. This is due to the scoring algorithm considering both the amount and the coherence of available evidence, using the composition and specialisation relationships. Figure 63 shows a considerable increase in the number of potential conclusions, resulting from the range of indicators that have been added to the routine.

```
CB: Printing final conclusion list for this segment:
CB: Read  2
CB: Read:File  2:8
CB: Read:Database  2:2
CB: Read:Record  2:1
CB: Read:APSMasterFile  2:1
CB: Read:CAF  2:1
CB: Read:PaymentFile  2:4
CB: Read:APSRecord  2:1
CB: Print  1
CB: Print:Report  1:1
CB: Print:Record  1:1
CB: Print:APSRecord  1:1
CB: Update  1
CB: Update:File  1:8
CB: Update:Database  1:2
CB: Update:Record  1:1
CB: Update:APSMasterFile  1:1
CB: Update:CAF  1:1
CB: Update:PaymentFile  1:4
CB: Update:APSRecord  1:1
CB: Call  2
CB: Output  1
CB: Output:File  1:8
CB: Output:Report  1:1
CB: Output:Database  1:2
CB: Output:Record  1:1
CB: Output:APSMasterFile  1:1
CB: Output:CAF  1:1
CB: Output:PaymentFile  1:4
CB: Output:APSRecord  1:1
CB: *DA* Finding highest scoring conclusions. (DAR 1)
CB: 1 high scoring conclusions.
CB: High scoring conclusion is Read:File, score 10
CB: *DA* Removing specialisations. (DAR 2)
CB: Found clear winner.
Winning conclusion for this segment currently Read:File
CB: ** Post-Disambiguation Processing **
CB: *PDAP* Specialisation Required.
CB: Found specialisation
CB: Found specialisation
CB: Found specialisation
Winning conclusion for this segment currently Read:PaymentFile
CB: *PDAP* Checking Thresholds.
Storing current winning conclusion
```

Figure 63: Extract From HB-CAS Log for the Random "Noise" Example

It has been established that the scoring algorithm can cope with situations where the majority of evidence is incoherent and unrelated to the correct concept. The disambiguation rules' ability to deal with unrelated but coherent indicators is now examined. Each rule is considered and justified in the context of the example routine.

8.4.1 Rule 1: Select Highest Scoring Conclusions

The effect of this rule can be seen in Figure 61 and Figure 63 where the Read:File concept scores higher than any other, and is selected for further processing. This

rule requires little justification since it is the basis of discriminating between conclusions.

8.4.2 Rule 2: Remove Specialisations

This rule aims to prevent various specialisations of an object concept competing with each other when they should be competing against a fundamentally different concept. In most cases, it has no effect because the scoring algorithm allocates points to the general forms of a specialised concept. These are in addition to points gained from the general concepts' own indicators. The general versions thus gain a greater score and are picked by rule 1 in preference to their specialisations. Should a general form win, forcing specialisation can retrieve the specialised version.

The rule is useful in situations where there is no *direct* evidence of the general concept, with the result that the general and specialised concept scores are identical. In this case, rule 2 ensures that if an arbitrary decision is ultimately required, the general form is not picked in favour of the specialisation. The arbitrary decision is made between the most general forms of competing concepts rather than versions of the same one. Figure 64 shows the original routine modified to trigger rule 2.

```
GD25   S10-READ SECTION.
NOISE *    PRINT A REPORT
NOISE *    READ
GD25   S10-000.
GD25       CALL 'GBFDAMA0' USING PAYMENT.
NOISE      CALL 'PRINT' USING P-PRINT.
GD25   S10-999.
GD25       EXIT.
```

Figure 64: Routine Modified to Demonstrate Rule 2

There is no direct evidence for the File concept so all of its score will come from its specialisation: PaymentFile. A fragment of the resulting assignment log is shown in Figure 65.

```
CB: Printing final conclusion list for this segment:
CB: Read  2
CB: Read:File  2:1
CB: Read:PaymentFile  2:1
CB: Print  2
CB: Print:Report  2:1
CB: Call  2
CB: *DA* Finding highest scoring conclusions. (DAR 1)
CB: 3 high scoring conclusions.
CB: High scoring conclusion is Read:File, score 3
CB: High scoring conclusion is Read:PaymentFile, score 3
CB: High scoring conclusion is Print:Report, score 3
CB: *DA* Removing specialisations. (DAR 2)
  : Attempting to generalise: 84
  : 1 generalisation found: 15
  : Attempting to generalise: 15
  : Error: 0 generalisations found, skipping...
  : Attempting to generalise: 84
  : 1 generalisation found: 15
  : Attempting to generalise: 15
  : Error: 0 generalisations found, skipping...
CB: Removing Read:PaymentFile
CB: *DA* Applying DAR 3.
CB: No clear winner, favouring composites over singles...
CB: *DA* Applying DAR 4.
CB: Still no clear winner, using highest action score...
CB: *DA* Applying DAR 5.
CB: Checking for same action in all composites...
CB: No further disambiguation possible, picking first conclusion
as winner.
Winning conclusion for this segment currently Read:File
CB: ** Post-Disambiguation Processing **
CB: *PDAP* Specialisation Required.
CB: Found specialisation
Winning conclusion for this segment currently Read:PaymentFile
CB: *PDAP* Checking Thresholds.
Storing current winning conclusion
```

Figure 65: Extract From HB-CAS Log Showing the Action of Rule 2

Given three equally high scoring conclusions, the specialised Read:PaymentFile concept is removed to allow Read:File and Print:Report to compete. An arbitrary decision was ultimately required and had the specialisation still been in contention, it may have lost to its general version. Removing and then re-introducing it later preserves the maximum amount of information.

This rule also may be required if the only winning concepts are the general version and specialisation. Rule 2 protects the information content of the specialisation by allowing the general form to win, without the need for an arbitrary choice between the two that the specialisation may lose.

8.4.3 Rule 3: Favour Composites over Non-Composites

This rule ensures that maximal information is provided to the user. When two equally high scoring concepts are winning, by favouring the composite over the

non-composite, information about an action and object is retained. Consider the example in Figure 66.

```
GD25   S10-READ-PAYMENT SECTION.
GD25   S10-000.
GD25       CALL 'GBFDAMA0' USING PAYMENT-FILE.
NOISE      CALL 'GBFDAMA0' USING DUMMY.
NOISE      CALL 'GBFDAMA0' USING DUMMY.
NOISE      CALL 'GBFDAMA0' USING DUMMY.
GD25   S10-999.
GD25       EXIT.
```

Figure 66: Routine Modified to Demonstrate Rule 3

In this case, a possible conclusion would be Call since there are a large number of calls in the routine. This does not convey as much information about the computational intent of the routine as the actual winner: Read:PaymentFile. The log extract shown in Figure 67 demonstrates the action of rule 3.

```
CB: Printing final conclusion list for this segment:
CB: Read  1
CB: Read:File  1:3
CB: Read:PaymentFile  1:2
CB: Call  4
CB: *DA* Finding highest scoring conclusions. (DAR 1)
CB: 2 high scoring conclusions.
CB: High scoring conclusion is Read:File, score 4
CB: High scoring conclusion is Call,score 4
CB: *DA* Removing specialisations. (DAR 2)
CB: *DA* Applying DAR 3.
CB: No clear winner, favouring composites over singles...
CB: Rejecting Call
CB: Single composite favoured over actions, selecting as winner.
Winning conclusion for this segment currently Read:File
CB: ** Post-Disambiguation Processing **
CB: *PDAP* Specialisation Required.
CB: Found specialisation
Winning conclusion for this segment currently Read:PaymentFile
CB: *PDAP* Checking Thresholds.
Storing current winning conclusion
```

Figure 67: Extract From HB-CAS Log Showing the Action of Rule 3

In this example, rule 3 has allowed higher quality information to be preserved. The rule also can be justified on the basis that composites contain a greater spread of evidence than non-composites. This implies greater coherence in the evidence since there is a probable relationship between the action and the object, in addition to the two entities existing independently.

8.4.4 Rule 4: Find the Highest Action Scores

Although non-composite actions have been removed from the list by this point, action concepts are still favoured over objects as the aim is to determine the computational intent. Rule 4 examines the action scores of composites. Those with the highest scores win.

The example routine was modified to trigger rule 4, as shown in Figure 68.

```
GD25    S10-READ-PAYMENT SECTION.
NOISE *     PRINT A PAYMENT REPORT
GD25    S10-000.
GD25        CALL 'GBFDAMA0' USING PAYMENT.
NOISE       MOVE REPORT TO P-PRINT.
GD25    S10-999.
GD25        EXIT.
```

Figure 68: Routine Modified to Demonstrate Rule 4

In this case it could be argued that the routine is becoming extremely ambiguous with either Read:PaymentFile or Print:Report being correct. This is an example of a coherent "interfering signal". As Figure 69 shows, Print:Report is selected because of the higher score of its action component.

```
CB: Printing final conclusion list for this segment:
CB: Read  1
CB: Read:File  1:3
CB: Read:PaymentFile  1:3
CB: Print  2
CB: Print:Report  2:2
CB: Call  1
CB: *DA* Finding highest scoring conclusions. (DAR 1)
CB: 3 high scoring conclusions.
CB: High scoring conclusion is Read:File, score 4
CB: High scoring conclusion is Read:PaymentFile, score 4
CB: High scoring conclusion is Print:Report, score 4
CB: *DA* Removing specialisations. (DAR 2)
  : Attempting to generalise: 84
  : 1 generalisation found: 15
  : Attempting to generalise: 15
  : Error: 0 generalisations found, skipping...
  : Attempting to generalise: 84
  : 1 generalisation found: 15
  : Attempting to generalise: 15
  : Error: 0 generalisations found, skipping...
CB: Removing Read:PaymentFile
CB: *DA* Applying DAR 3.
CB: No clear winner, favouring composites over singles...
CB: *DA* Applying DAR 4.
CB: Still no clear winner, using highest action score...
CB: Rejecting Read
CB: Single highest action score found, selecting as winner.
Winning conclusion for this segment currently Print:Report
CB: ** Post-Disambiguation Processing **
CB: *PDAP* Specialisation Required.
Winning conclusion for this segment currently Print:Report
CB: *PDAP* Checking Thresholds.
Storing current winning conclusion
```

Figure 69: Extract from HB-CAS Log Showing the Action of Rule 4

8.4.5 Rule 5: Common Action Component

In the event that there are still multiple winners, rule 5 allows conflicts between general forms of object concepts in composites to be handled gracefully. It achieves this by removing the objects from the composites to leave the action concept common to all of them. Figure 70 shows the original routine modified to include indicators for reading a database, in addition to those for reading the file.

```
GD25   S10-READ-PAYMENT SECTION.
NOISE *    CIF ACCESS MADE
GD25   S10-000.
GD25       CALL 'GBFDAMA0' USING PAYMENT.
NOISE      MOVE SPACES TO CIF-PARMS.
GD25   S10-999.
GD25       EXIT.
```

Figure 70: Routine Modified to Demonstrate Rule 5

The resulting log is shown in Figure 71.

```
CB: Printing final conclusion list for this segment:
CB: Read  1
CB: Read:File  1:2
CB: Read:Database  1:2
CB: Read:PaymentFile  1:2
CB: Read:CMS  1:2
CB: Call  1
CB: *DA* Finding highest scoring conclusions. (DAR 1)
CB: 4 high scoring conclusions.
CB: High scoring conclusion is Read:File, score 3
CB: High scoring conclusion is Read:Database, score 3
CB: High scoring conclusion is Read:PaymentFile, score 3
CB: High scoring conclusion is Read:CMS, score 3
CB: *DA* Removing specialisations. (DAR 2)
.
.
.
CB: *DA* Applying DAR 3.
CB: No clear winner, favouring composites over singles...
CB: *DA* Applying DAR 4.
CB: Still no clear winner, using highest action score...
CB: *DA* Applying DAR 5.
CB: Checking for same action in all composites...
CB: Same action found, selecting as winner.
Winning conclusion for this segment currently Read
CB: ** Post-Disambiguation Processing **
CB: *PDAP* Specialisation Required.
Winning conclusion for this segment currently Read
CB: *PDAP* Checking Thresholds.
Storing current winning conclusion
```

Figure 71: Extract from HB-CAS Log Showing the Action of Rule 5

Read is selected as the winning conclusion because the object evidence is contradictory. Note that despite *forced_specialisation* being set to True, HB-CA cannot specialise in situations like this where the object evidence is ambiguous. This is because no objects exist in the ultimate winner; it is a single action concept.

In the event that the rules fail to solve the conflict between conclusions, an arbitrary decision is made and the first in the list is picked. This has the unintentional side effect of potentially improving concept assignment performance, as the first conclusion is likely to be derived from the subroutine name. This would not apply to SOM-created segments.

8.4.6 Post-Disambiguation Processing

This stage (presented in section 6.3.2.4) involves forcing the selection of the most specialised form of a concept for which there is evidence, and checking that the

winning concept scores above the required threshold. The latter issue is trivial and is not discussed further. Forced specialisation merits greater examination.

In general, forcing specialisation is a successful way of retrieving the most specialised form of a concept when its general version has won. The method by which this is performed selects all forms of the winning concept for which there is evidence and picks the one that scores highest. If more than one achieves the high score then the result is regarded as ambiguous and the general form is left as the winner.

The main problem with this ambiguity rule is its assumption that specialisation will occur in only one level (e.g. File to MasterFile, File to APSMasterFile). Additional methods would need to be defined to handle intermediate specialisations (e.g. File to MasterFile to APSMasterFile) because if an intermediate and most specialised form of a concept scored the same, the result would be regarded as ambiguous. In such situations, the intermediate should be picked (being a more general version of the specialised form) but the rule will actually choose the common general version of both (File in this example). Consequently, the rule will not produce incorrect results but some precision may be lost. This happens because both the intermediate and most specialised forms are regarded as being at the same level (the list used by the rule is a flattened form of the library's structured representation). If there is no ambiguity, the situation does not arise and the intermediate and most specialised form will compete normally.

Figure 72 and Figure 73 show the example routine and a log extract demonstrating a situation where forced specialisation cannot be performed due to a conflict between two specialised versions of a general concept. There are no intermediate concepts in the library shown in the Appendix so the ambiguity problem discussed above will not arise.

```
GD25    S10-READ-PAYMENT SECTION.
NOISE *     CAF ACCESS MADE
GD25    S10-000.
GD25        CALL 'GBFDAMA0' USING PAYMENT.
NOISE       MOVE SPACES TO CAF-PARMS.
GD25    S10-999.
GD25        EXIT.
```

Figure 72: Routine Modified to Demonstrate Forced Specialisation

```
CB: Printing final conclusion list for this segment:
CB: Read  1
CB: Read:File  1:4
CB: Read:CAF  1:2
CB: Read:PaymentFile  1:2
CB: Call  1
CB: *DA* Finding highest scoring conclusions. (DAR 1)
CB: 1 high scoring conclusions.
CB: High scoring conclusion is Read:File, score 5
CB: *DA* Removing specialisations. (DAR 2)
CB: Found clear winner.
Winning conclusion for this segment currently Read:File
CB: ** Post-Disambiguation Processing **
CB: *PDAP* Specialisation Required.
CB: Found specialisation
CB: Found specialisation
Winning conclusion for this segment currently Read:File
CB: *PDAP* Checking Thresholds.
Storing current winning conclusion
```

Figure 73: Extract From HB-CAS Log Showing the Forcing of Specialisation

8.4.7 Levels of Ambiguity

This section presents the results of an investigation into the frequency with which HB-CA's disambiguation rules are applied. The parameters for the investigation are shown in Table 14. In each case, the HB-CAS log was examined to determine how often the various rules had been invoked.

Program Set	Set D
Library Content	Appendix, Section A.2
rec_tbresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Section A.4.1

Table 14: Parameters for Investigation of Disambiguation Rule Triggering

Figure 74 shows the relative proportions of rule triggering.

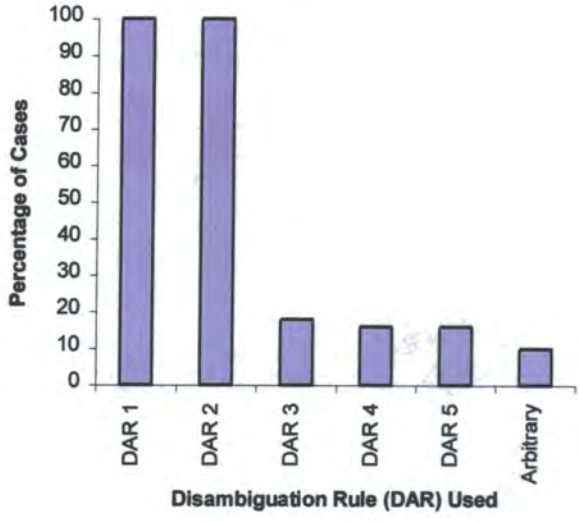


Figure 74: Chart to show the Proportion of Cases in which Disambiguation Rules are Triggered

As might be expected, rules 1 and 2 are triggered in every case since they are required for basic discrimination. It is interesting to note that none of the cases tested showed rule 4 disambiguating more successfully than rule 5. The low percentage of arbitrarily decided cases confirms the effectiveness of the rules.

8.4.8 Summary

This section has discussed the ability of HB-CA to deal with ambiguity in the evidence presented. The properties of the scoring algorithm and disambiguation rules have been considered with reference to an example code fragment from a real program. This was modified to demonstrate situations where the rules apply.

There is reasonable justification for the rules used, given the general aims and priorities of the HB-CA method. Should these aims change, the disambiguation criteria may also require modification. HB-CA has proved to be capable of gracefully degrading its concept assignment performance, with relatively few cases decided arbitrarily.

8.5 Library Content

In view of HB-CA's sensitivity to the library, this section briefly discusses some issues to be considered when creating its content. These are drawn from theoretical aspects of its structure and experience gained from undertaking the investigations in this chapter. The representational power of the library is discussed in section 8.9.

Although it is impossible to suggest what should be the optimal content of the library in a particular instance, some general principles have emerged from using HB-CAS for the investigations in this chapter. Indicators ideally need to be unique to a particular concept. There are exceptions to this (e.g. most files will need a file indicator, regardless of their specific nature) but unique indicators improve the quality of hypothesis generation and consequently cause less confusion in concept binding. It is suggested that secondary concepts should be allocated the indicators for their more general versions, in addition to their own specific and differentiating indicators. This provides for successful recognition when there is no direct evidence of the general concept.

The representational power of the method is discussed in section 8.9, but at this point, it is worth noting the different ways in which evidence for a concept can be assembled. The set of indicators for a concept of low-level abstraction is likely to be similar to the concept name, e.g. MasterFile will have indicators such as "File" and "Master" in various classes. As the level of abstraction rises, a different approach may be required as routines that implement more functionality are likely to call on lower-level subroutines to do the work. Consequently, the indicators for a high-level concept will be the subroutine names as found in the calling statements. When creating high-level concepts in the library the indicators therefore should be related to the constituent parts of the solution rather than the name of the solution itself, as the evidence in the code will be diverse rather than coherent.

This section has briefly discussed some practical considerations for creating library content. These include the uniqueness of indicators and the body of evidence for a concept.

8.6 Computational Cost

This section identifies several factors that have a significant impact on the computational cost of HB-CA. Biggerstaff et al. claim that plausible reasoning systems (like HB-CA) appear to have linear computational growth with the length of program under analysis [BIGG93]. It is expected that HB-CA will exhibit this cost characteristic.

A large proportion of the HB-CA process involves comparing source code to the library, and these entities have the biggest impact on its computational cost. The discussion in this section focuses on structural attributes of both.

8.6.1 Source Code

It is important to consider the impact of the source code being analysed because it is likely to change more frequently than any other entity involved in HB-CA. The two characteristics that have the greatest effect on HB-CA's computational cost are the source code length, and the number of sections. HB-CA should have linear computational growth with the length of source code under analysis.

8.6.1.1 Source Code Length

The source code is an essentially linear structure (when treated as a body of text by HB-CA) and as such, it is reasonable to expect that the computational cost of HB-CA should increase linearly with the length of source code being analysed. This assumes that the library being used remains constant.

To verify this relationship an investigation has been undertaken. The execution time of a module or part-module is regarded as directly proportional to the computational cost of the method it implements. Consequently, the discussion in this section uses cost and execution time synonymously. The modules of HB-CAS supply these timings, accurate to within 1 second.

The source programs were selected semi-randomly from a set of 150. The selection criteria were to include the shortest and longest available programs, space the program lengths by approximately 50 lines, and ensure that programs were drawn

from the same system. HB-CA's performance on files from a different system is discussed in section 8.10.

The parameters for the investigation are shown in Table 15.

Program Set	Set E
Library Content	Chapter 3, Section 3.7.2
rec_thresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Sections A.5.1, A.5.2, A.5.3, A.5.4

Table 15: Parameters for Investigation of Computational Cost

Using the results shown in section A.5.1, the relationship between the program length and the computational cost of HB-CA is presented in Figure 75.

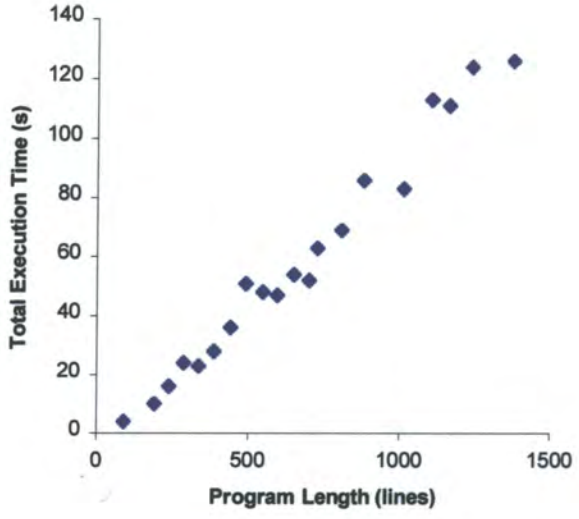


Figure 75: Graph to show the relationship between the Total Execution Time and Program Length

As expected, there is a clear linear relationship between the properties. It is interesting to investigate what proportion of the total cost is provided by each stage of HB-CA. This is shown in Figure 76.

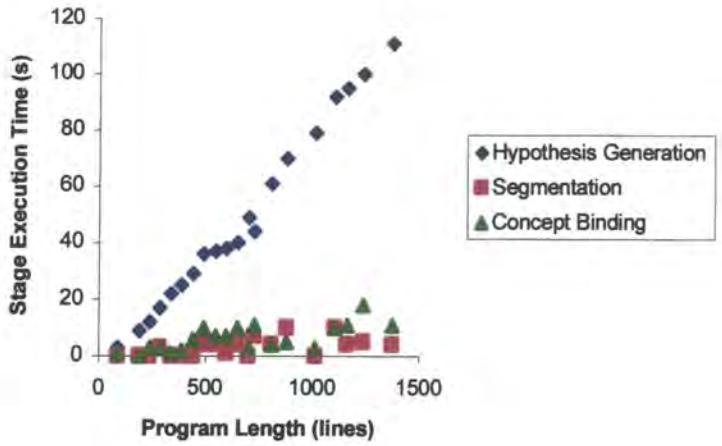


Figure 76: Graph to show the relationship between the Stage Execution Time and Program Length

It is clear that all three costs rise in an approximately linear manner with increasing program length. Segmentation and concept binding do not show this relationship as clearly as hypothesis generation, but any variation in their individual costs will not be evident clearly in the total as hypothesis generation dominates the overall cost. If less indicator classes are used, these variations might be more apparent as the total cost would be more sensitive to the segmentation and concept binding stages.

This section has established the expected linear relationship between HB-CA's overall computational cost and the length of the program being analysed. The next section investigates the effect of source code length on those stages of HB-CA that use it directly.

8.6.1.2 Direct Effects of Source Code Length

The hypothesis generation stage of HB-CA transforms the source code into a hypothesis list, by comparing the indicators in the library with tokens extracted from the source code. This process is termed indicator recognition and is presented in section 4.3. It has two parts: extraction, and matching. The second process that may be involved in hypothesis generation is sorting. The necessity of a sort algorithm is implementation-dependent since indicator recognition could be implemented to store the hypotheses in order initially. Consequently, the cost of

sorting is not addressed here. Hypothesis generation cost is regarded therefore as directly proportional to indicator recognition cost.

Assuming an even density of tokens per line, the number of tokens extracted from the source code should be approximately proportional to the length of the program (measured in lines). Figure 77 shows the results of the investigation outlined in Table 15, verifying this relationship.

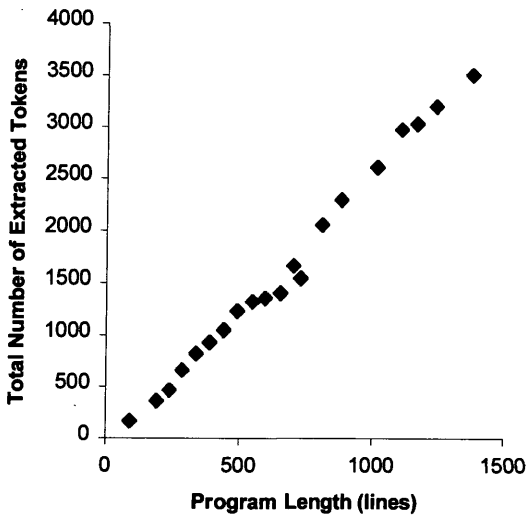


Figure 77: Graph to show the relationship between the Total Number of Extracted Tokens and Program Length

Since the number of extracted tokens is linearly related to the program length, and the next stage of hypothesis generation (matching) involves comparing extracted tokens with stored indicators, it is reasonable to expect the computational cost of matching also to be related linearly to the program length. This is because the matching algorithm compares each token to every indicator in the library. The use of flexible matching options proportionally increases the cost since every option used incurs an additional test on each library indicator. The exception is case sensitivity, which is employed in the first test if necessary. Synonym matching incurs the additional test and multiplies the comparison cost by the number of synonyms found for a particular token. The size of the synonym list is static and consequently there is a fixed upper limit on this cost for any particular token. This means that there should be no fundamental effect on the linear relationship between

the computational cost of the indicator matching process and the length of the source code.

The only options used for the investigations in this chapter were case insensitivity and sub-string matching for modules that implement them. Experiments carried out during the development of HB-CA showed this set of options to be the most successful for indicator recognition. Synonym matching was found to reduce indicator recognition accuracy significantly, in addition to substantially increasing the cost of execution.

Further results from the investigation described in Table 15 are shown in Figure 78, appearing to confirm linear computational growth with source code length for indicator recognition.

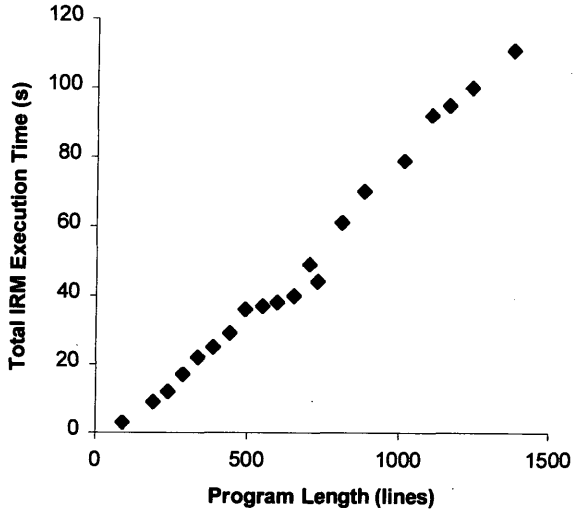


Figure 78: Graph to show the relationship between the Total IRM Execution Time and Program Length

It is also interesting to compare the execution times of the individual indicator recognition modules to determine whether there is any significant difference between them. The level of matching varies widely between the modules, with keyword recognition matching indicators in only two of the twenty programs, despite extracting more tokens than any other module.

Figure 79 shows the individual execution times for the modules. Results are drawn from the investigation described in Table 15. Segment boundary matching is not shown, as its execution times were negligible.

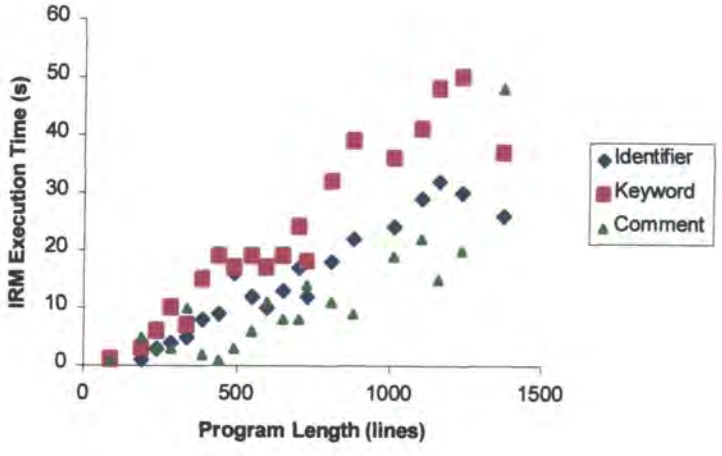


Figure 79: Graph to show the relationship between the Individual IRM Execution Times and Program Length

When separated, the execution times are not as clearly linear as their sum and the respective gradients of the cost-increase for each module differ slightly. This could be interpreted to mean that the individual modules respond differently given different program lengths, or simply that the proportion of each type of token differs between programs in the data set. The latter would require the modules to do different amounts of work. This can be determined by comparing the proportion of the total time taken by each module with the proportion of tokens it extracts in each case. This is shown in Figure 80 using data from the investigation described in Table 15.

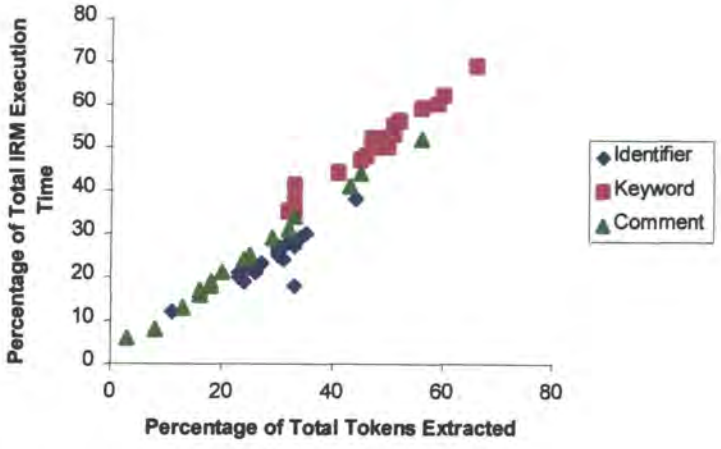


Figure 80: Graph to show the relationship between the Proportion of Total IRM Execution Time and the Proportion of Total Tokens Extracted for Each IRM

Since the relationship between the proportions is clearly linear, the slight discrepancies shown in Figure 79 can be attributed to the characteristics of the source code under analysis rather than differences in the recognition modules and the methods underlying them.

In summary, theoretical analysis of the algorithms in indicator recognition indicates that the computational cost should grow linearly with the length of the source code given a fixed library. Applying HB-CAS to a number of programs indicates that this is the case and hence the hypothesis generation cost also varies linearly with the length of the source code.

The source code is not used by any other part of HB-CA and consequently should not have a direct length-related effect on the computational cost of segmentation or concept binding. Longer programs are likely to contain more information and thus increase the cost of these stages, but there is no direct link with program size. Another source code characteristic that can influence the computational cost of HB-CA is the number of sections.

8.6.1.3 Direct Effects of the Number of Sections

The number of sections has a minimal impact on hypothesis generation since it only affects the production of segment boundary hypotheses. There are so few segment

boundaries compared to the other token types in a program that the cost of matching them in hypothesis generation is negligible. The number of sections also has no direct effect on concept binding since the intervening segmentation stage may subdivide the original sections. A direct relationship therefore cannot be shown.

Segmentation is affected by the number of sections, as it is performed initially using the subroutine boundaries in the COBOL II program being analysed. The computational cost of processing a single section is incurred for each. Consequently, it is reasonable to expect the computational cost of executing the segmentation algorithms to vary linearly with the number of sections in the source code. Figure 81 shows more data from the investigation described in Table 15, presenting the relationship between the segmentation time and the number of sections.

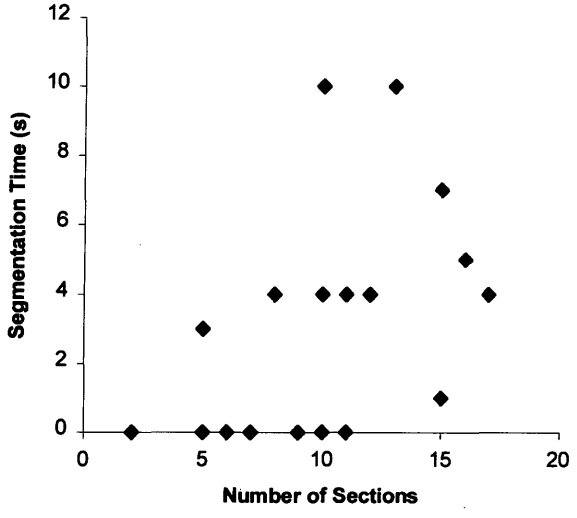


Figure 81: Graph to show the relationship between the Segmentation Time and the Number of Sections in the Source Code (Low Resolution Timers)

This graph does not demonstrate a clear linear relationship. Nonetheless, cost differences are apparent and further analysis must be undertaken to explain them.

Large changes in cost could be explained by the use of SOMs in the segmentation process. SOMs would normally be required in large programs with few sections,

assuming an even distribution of indicators in the program population. Consequently, the computational cost of segmentation could be affected if a SOM is required for clustering. Using a SOM would increase the segmentation cost by a fixed amount, plus a variable amount linearly related to the number of hypotheses in the segment being considered. Figure 82 shows the correlation between the time taken for segmentation, and the number of SOMs used during the process (data is drawn from the investigation described in Table 15).

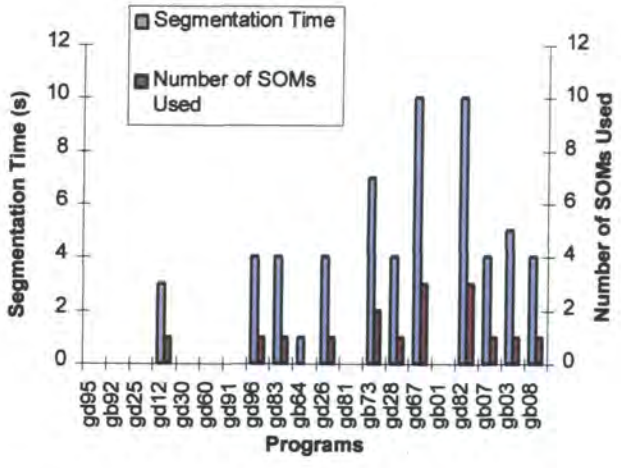


Figure 82: Chart to compare the Segmentation Time and the Number of SOMs Used for Various Programs

As expected, there is a "step" increase in the cost for every use of a SOM during segmentation. This increase can be illustrated more clearly by plotting the two data series against each other as shown in Figure 83.

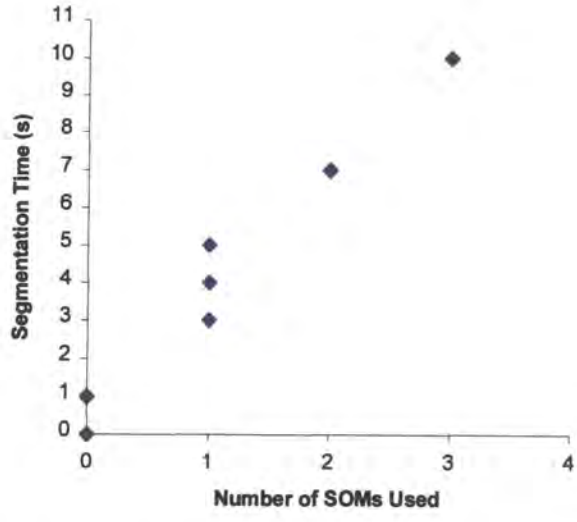


Figure 83: Graph to show the relationship between the Segmentation Time and the Number of SOMs Used

The original, low-resolution timers built into HB-CAS cannot detect the smaller changes in cost that are linked to the number of sections, but only the large changes caused by SOM usage. The system was modified to include timers capable of millisecond resolution and the investigation repeated. The expected, approximately linear relationship between the segmentation time and the number of sections now can be seen, particularly if the results are separated by the number of SOMs used. This is shown in Figure 84.

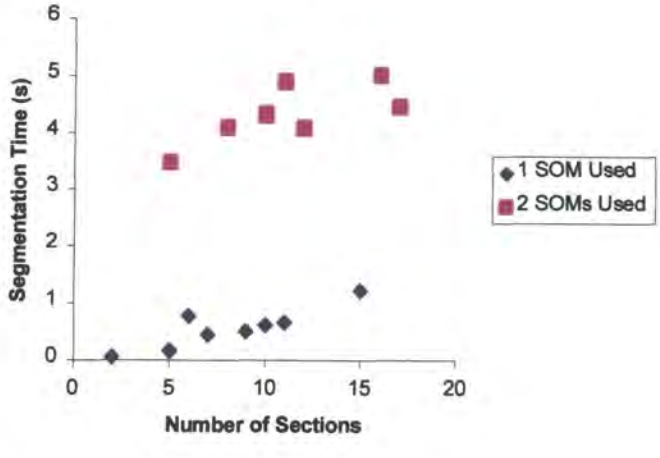


Figure 84: Graph to show the relationship between the Segmentation Time and the Number of Sections in the Source Code (High Resolution Timers)

8.6.1.4 Summary

Section 8.6.1 has established the characteristics of the source code that have a major impact on the computational cost of HB-CA. Section 8.6.1.1 established that the overall computational cost of HB-CA is related linearly to the length of program being analysed. Section 8.6.1.2 demonstrated a linear relationship between the length of the source code and the hypothesis generation cost. Section 8.6.1.3 showed that the cost of segmentation is related linearly to the number of sections in the source code (when SOM-related costs are ignored).

8.6.2 Library

Since the library is used in most stages of HB-CA, it is important to consider its impact on the computational cost of the process. The library is a non-linear structure and consequently it is not reasonable to expect the computational cost of HB-CA to vary linearly with changes in its content and size.

To investigate the library's impact on the computational cost, it is examined with reference to some of its constituent entities and relationships.

8.6.2.1 The Library in Hypothesis Generation

This section discusses the relationship between the computational cost of hypothesis generation and the number of indicators and indicates relationships.

The parts of the library used in hypothesis generation are indicators, concepts, and the indicates relationship. Section 8.6.1.2 demonstrated that hypothesis generation has linear computational cost variation in relation to the length of source code being analysed. Modifying the library should not cause a change in this relationship although additional content will necessarily cost more to use.

Indicator matching involves comparing each extracted token with every indicator in the library (effectively comparing two lists). An increase in the number of indicators should result in a linear increase in the execution time, signifying a similar variation in the computational cost of the method. This linear increase must also be considered with reference to the number of concepts to which an indicator is linked. The cost of matching a single indicator is multiplied by the cost of

producing hypotheses for its concepts. Doubling the number of indicators in the library should double the cost of recognition. Doubling instances of the indicates relationship for the original set of indicators should have a similar effect. Either change alone should have a linear effect on the cost of hypothesis generation, but if made together they may produce a different characteristic.

An investigation has been undertaken to gain some validation of these claims. The largest source file from Set E was used to ensure any differences in the results were as clear as possible. The library content presented in Chapter 3 was used for the first execution. Subsequently, one indicator was removed from the library for each execution, until none remained. Indicator removal was distributed evenly among the concepts, where possible in the order: keyword, comment, identifier. Investigation parameters are shown in Table 16.

Program Set	Set F
Library Content	Chapter 3, Section 3.7.2
rec_thresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Section A.5.5

Table 16: Parameters for Investigation of Indicator Cost

Initial investigations revealed that the variable proportion of the execution time, i.e. that which is dependent on the size of the library, was negligible in comparison to the fixed cost of execution. Consequently, no clear relationship was apparent with low-resolution timers. The high-resolution timers produced the results shown in Figure 85.

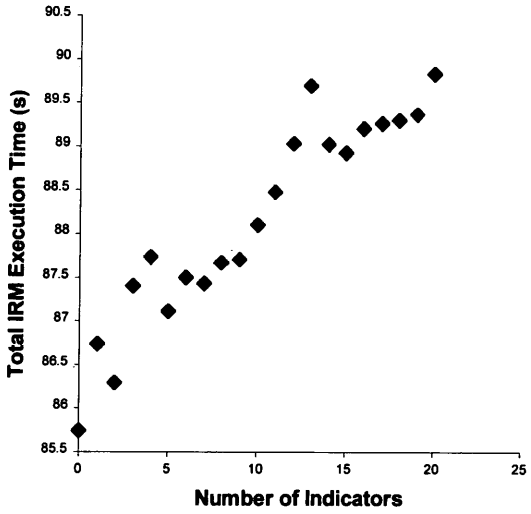


Figure 85: Graph to show the relationship between the Total IRM Execution Time and the Number of Library Indicators

These timers make the results more sensitive to fluctuations in the operating system, but the approximately linear characteristic of the relationship is quite clear.

In summary, the computational cost of hypothesis generation varies linearly with the number of indicators in the library.

The effect of modifying the indicates relationship should be similar to that of modifying the number of indicators in the library. Hypothesis generation cost is dependent on the number of matched indicators and the number of concepts to which each is linked. Increasing the number of links (i.e. instances of the indicates relationship) should increase the time taken to produce hypotheses in a linear manner.

An investigation was undertaken using the library content presented in Chapter 3 and the largest source program from Set E. It begins with 23 instances of the indicates relationship, and each successive execution of the system adds an additional instance to approximately one third of the indicators. This results in confusing and incorrect concept assignment but the concern here is the change in cost, not the accuracy of the result. Investigation parameters are shown in Table 17.

Program Set	Set F
Library Content	Chapter 3, Section 3.7.2
rec_thresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Section A.5.6

Table 17: Parameters for Investigation of Indicates Cost

The relationship between total indicator recognition time and instances of indicates is shown in Figure 86.

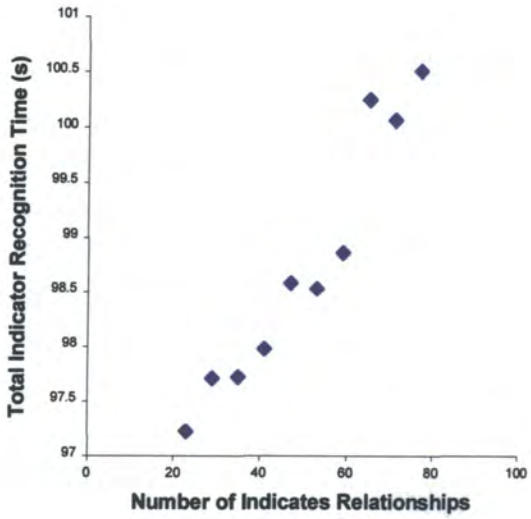


Figure 86: Graph to show the relationship between the Total Indicator Recognition Time and the Number of Indicates Relationships

The expected linear relationship is evident.

Adding instances of indicates to existing concepts and indicators, has the same effect on indicator recognition costs as adding new concepts to the library and linking indicators to them. It has a similar structural effect (i.e. more instances of indicates).

In summary, the computational cost of hypothesis generation varies linearly with the number of instances of the indicates relationship.

8.6.2.2 The Library in Segmentation and Concept Binding

The library has minimal effect on segmentation as the process operates on the hypothesis list only.

The library is used directly in concept binding and consequently has an effect on the computational cost of this stage of HB-CA. The two salient relationships are composition and specialisation. Indicates is not used in concept binding.

The cost of concept binding should vary linearly with the number of segments since the same section of processing must be performed once for each. Using the investigation shown in Table 15 (with the lower resolution timers), Figure 87 shows some validation of this relationship. The results are shown in section A.5.1.

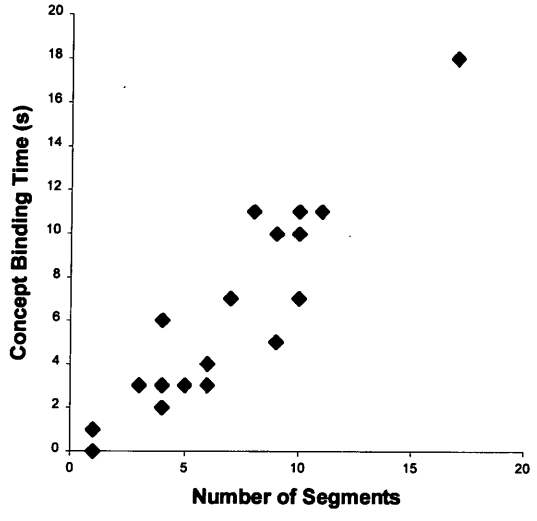


Figure 87: Graph to show the relationship between the Concept Binding Time and the Number of Segments

There are four activities within concept binding and these are described in Chapter 6. Factors in the cost of each activity are now considered.

8.6.2.3 Factors in Conclusion Generation Cost

The first concept binding activity, conclusion generation, involves generating all possible conclusions from the action concepts in a segment. Both composition and specialisation are used in this. Each action concept generates conclusions based on its composites. Conclusions are also generated for every specialisation of a composite object. Changes in the number of segments, the number of compositions, or the number of specialisations, may all have an effect on the cost of conclusion generation. These are all different types of change and are considered separately.

Conclusion generation has a linear relationship with the number of action concepts in the current segment because each concept generates one set of conclusions. This holds as long as each set of conclusions is regarded as a single unit of cost. However, once the composition and specialisation relationships are considered, the cost relationship is not so clearly defined.

In a library with no specialisations, adding composites will vary the conclusion generation cost linearly, since each additional composite will generate a single additional conclusion for a particular action concept.

In a library with a fixed number of composites, adding specialisations will cause a linear variation in cost, if every action concept is composite with the primary object being specialised. This is because one additional conclusion will be generated for each action concept. Those action concepts that are not composite with the primary object being specialised will remain unaffected by the change. Cost variation is linear for further additional specialisations of a particular primary object, but if another object is specialised, the overall characteristic may be non-linear. This can be illustrated by the following example: assume that Read is composite with File and Record, and Write is composite with Record. This would produce 5 conclusions (Read, Read:File, Read:Record, Write, Write:Record). By specialising File to MasterFile, the number of conclusions would increase to 6 because the Read concept has a composite relationship with File. Specialising File again to PaymentFile would result in 7 conclusions. If Record were then specialised, the number of conclusions would increase to 9 because both action concepts are

affected. The cost characteristic is therefore linear within a single group of specialisations (e.g. specialisations of File), but non-linear in general.

Non-linear variation also may be seen if a new composite is created in a library with specialisations. In this case, the new composite will increase the cost in proportion to the number of specialisations attached to its object concept.

A small investigation to verify some of these arguments was carried out using the largest program from Set E, and gradually extending the library content presented in Chapter 3. Cost variation was measured in terms of the execution time of the conclusion generation subroutine in the concept assignment module of HB-CAS. Times for all segments in a program were accumulated to give the total shown in the results. The library was extended by adding concepts with no indicators, thus ensuring that only the composition and specialisation relationships were analysed. For the first part of the investigation, sets of specialisations were added to each of the two primary objects in the library. There are 4 composites in the library and these were not modified. The investigation parameters are shown in Table 18.

Program Set	Set F
Library Content	Chapter 3, Section 3.7.2
rec_thresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Sections A.5.7, A.6

Table 18: Parameters for Investigation of Specialisation/Composition Cost

Figure 88 shows the change in cost being linear for each set of specialisations shown. The change in gradient indicates when the object concept, to which specialisations were added, was changed. This shows that the cost change is linear within a particular group of specialisations, i.e. specialisations of one object concept. Changing the root of the group (i.e. the object concept) causes a change in the gradient but the cost still has a linear relationship with the size of the group being considered.

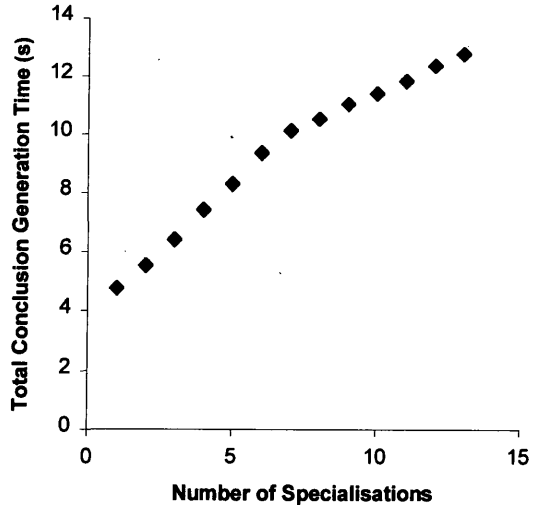


Figure 88: Graph to show the relationship between the Total Conclusion Generation Time and the Number of Specialisations in the Library

The effect of adding composites was measured by adding additional object concepts to the library and creating each as a composite. Compositions were made with two action concepts and, although the results are not as clear as those for specialisation, there is a change in gradient where the action concept being used was changed (see Figure 89, with results in section A.5.7.2).

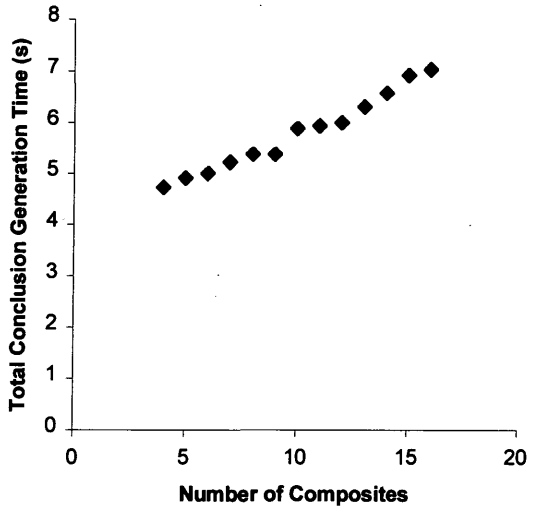


Figure 89: Graph to show the relationship between the Total Conclusion Generation Time and the Number of Composites in the Library

In summary, the cost of conclusion generation can be related linearly to various aspects of the library, but the overall relationship is determined by a combination of these factors.

8.6.2.4 Factors in Conclusion Completion Cost

The second stage of concept binding is conclusion completion. This involves using the object concepts in the current segment to validate composite conclusions in the list. The only library relationship used is specialisation, to determine the more general forms of a particular specialised object concept. Changes in the specialisation relationship could have, at worst, a linear effect on the cost of this process. This is because one additional instance of the relationship may need to be examined for every additional specialisation added. The worst case would arise if every conclusion contained a form of the specialisation, and the most specialised form was hypothesised in every case. Every hypothesis would then add score to every conclusion. This is unlikely to happen in practice and hence changes to the specialisation relationship should not have a major impact on the cost of conclusion completion.

8.6.2.5 Factors in Disambiguation

The third part of concept binding, disambiguation, also uses specialisation in a similar way to conclusion completion. The relationship is used to find general forms of concepts where the evidence for specialised forms is ambiguous. The addition of a specialisation to the library may have therefore a similar effect to that in conclusion completion. In the worst case, where every situation requires the examination of all forms of a concept, the change in computational cost would be linear, but in practice, it should be better than this because it is unlikely that every case will require this processing.

8.6.2.6 Factors in Post-Disambiguation Processing

The final stage of concept binding involves specialising a general concept if *forced_specialisation* is True. In the worst case, this will require examination of every specialisation of that concept and consequently, the addition of a new specialisation to the library could cause a linear change in the cost. This change would be

observed if the new specialisation was the most specialised form of the general concept, and there was evidence for it. In practice, it is unlikely that this would occur in every segment analysed.

8.6.2.7 Summary

In summary, the major factors in the computational cost of concept binding are the numbers of composition and specialisation relationships in the library. Independent changes in these relationships are not expected to cause anything worse than linear change in the computational cost. When changes are made to both relationships simultaneously, or when a single change effectively results in this, e.g. adding a composite with specialised objects as discussed above, the change in computational cost may not be linear.

8.6.3 Summary

The effect of various properties of the source code and library on HB-CA's computational cost has been considered. HB-CA possesses a key characteristic of plausible reasoning systems: linear growth in computational cost with the length of the source code being analysed. Cost factors other than source code length have been identified, and their impact on the computational cost of individual parts of HB-CA considered.

8.7 Spatial Cost

This section considers the spatial cost of HB-CA.

HB-CA's spatial cost is closely linked to its computational cost. The linear relationship between source code length and computational cost is reflected in the size of data structures created by the various stages of HB-CA.

8.7.1 Hypothesis Generation

Recall that the number of tokens extracted rises linearly with the length of source code (see Figure 77). The spatial cost of extraction increases at the same rate because each token is one element of the data structure produced by the extraction stage.

In the matching stage, the number of hypotheses represents the spatial cost. The number of hypotheses generated will be, at most, the number of tokens multiplied by the number of indicators in the library, multiplied by the number of indicates relationships in the library. This is the worst-case situation that would occur if every token was matched, and every indicator was linked to every concept. In practice, this is highly unlikely since the resulting hypothesis list almost certainly would be useless for concept assignment. Section 8.6.2.1 demonstrated a linear relationship between the computational cost and the numbers of indicators and indicates relationships in the library. A similar relationship holds for the spatial cost of hypothesis generation, due to the calculation shown above. If the number of indicators or indicates relationships is increased, the worst-case situation would result in a linear increase in the space required for the hypothesis list.

8.7.2 Segmentation

The segmentation spatial cost is mainly proportional to the size of the hypothesis list generated in the first stage, as it is the primary data structure used by this process. Additional costs may be incurred if a SOM is required, the worst-case spatial cost of this being related to the size of the largest segment in the hypothesis list. If there is only one segment (i.e. no subroutines in the source), the SOM can be no larger than the entire hypothesis list.

8.7.3 Concept Binding

The spatial cost of concept binding has similar dependencies to its computational cost. Changes in the library content, in terms of composition and specialisation, may cause, in the worst case, linear changes in the spatial cost. This is demonstrated for conclusion generation in Figure 90 and Figure 91 where the spatial cost (in terms of conclusions) is shown in relation to the numbers of specialisation and composition relationships in the library. The results are from the investigation described in Table 18 and show the total number of conclusions generated for the program in each case. Further details of the investigation can be found in section 8.6.2.3.

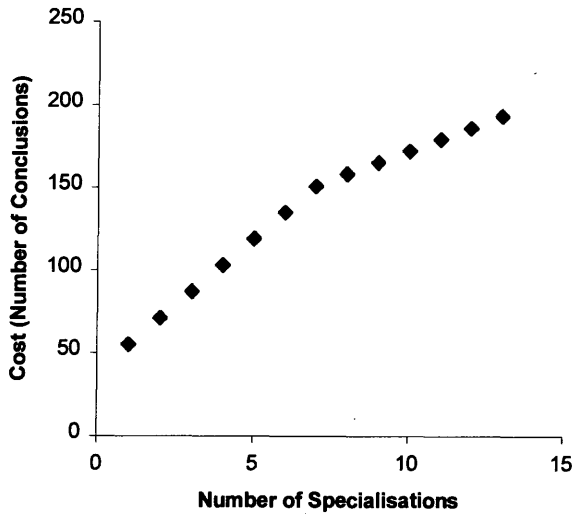


Figure 90: Graph to show the relationship between the Spatial Cost of Conclusion Generation and the Number of Specialisations

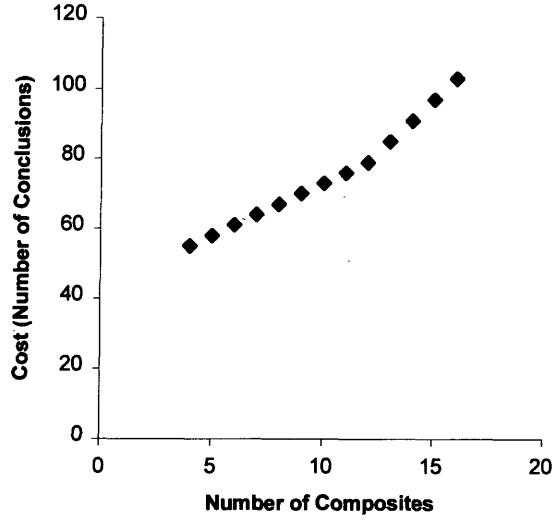


Figure 91: Graph to show the relationship between the Spatial Cost of Conclusion Generation and the Number of Composites

The change in gradient on both graphs results from the relationships being applied to different concepts. The results show a linear change for additions to each concept individually. It is interesting to note that the shapes of these graphs are similar to those shown in Figure 88 and Figure 89. This indicates that the number

of conclusions generated has a direct relationship with the computational cost of conclusion generation.

8.7.4 Library

The spatial cost of the library is, at worst, linear with the number of concepts or indicators represented. Adding either entity causes a linear change in the space required. This cost characteristic can be improved for concepts, using the composition and specialisation relationships. Adding an instance of either relationship causes a linear cost increase but, since there are implied compositions with the specialisations of a primary concept, the amount of information represented can be increased by more than one item. If every concept is atomic then the increase is linear, but if composition and specialisation are used it can be better than linear. This is one advantage of using a semantic network rather than a list to represent concepts.

8.8 Expandability

This section discusses the ability of HB-CA to incorporate different information sources for concept assignment.

A particular strength of HB-CA is its use of a source-code independent representation. By transforming the source code into a hypothesis list very early in the process, the latter two stages of HB-CA can use information of any type. This assumes that the information relates to the source code at the token or line level and is transformable into a hypothesis. The ability to use multiple information sources can be seen by considering the hypothesis generation stage of HB-CA. Each indicator recognition process generates a list of hypotheses that are merged to form a single, ordered list. Clearly, if an additional list is included, the extra information can be merged without difficulty.

The major issue to be considered when adding an additional source of information is whether the potential information gain is worth the cost of extraction.

HB-CA defines four classes of indicator (see Chapter 4). An investigation has been undertaken to determine their relative effectiveness. Indicator recognition modules

were executed separately, and in various combinations, to determine their effect on concept assignment. The investigation was simplified by selecting programs that do not require SOM analysis to ensure that each segment was processed using a subset of the same total indicator collection. The distribution of indicators among concepts in the library is reasonably even so there should be no particular bias towards one indicator class. The investigation parameters are shown in Table 19.

Program Set	Set G
Library Content	Appendix, Section A.2
rec_thresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Section A.7

Table 19: Parameters for Investigation of Expandability

The relative computational costs of indicator recognition are discussed in section 8.6.1.2.

Segment boundary indicators are treated differently to the other types. Although they are similar to the other classes and could be treated in a similar way, HB-CAS relies on the presence of at least one pair in the hypothesis list rather than using a SOM when none are available. Segment boundary recognition is executed therefore in all parts of the investigation.

Figure 92 shows the relative proportions of the "total" concept assignment achieved by each module or combination (all concepts were non-strictly accurate).

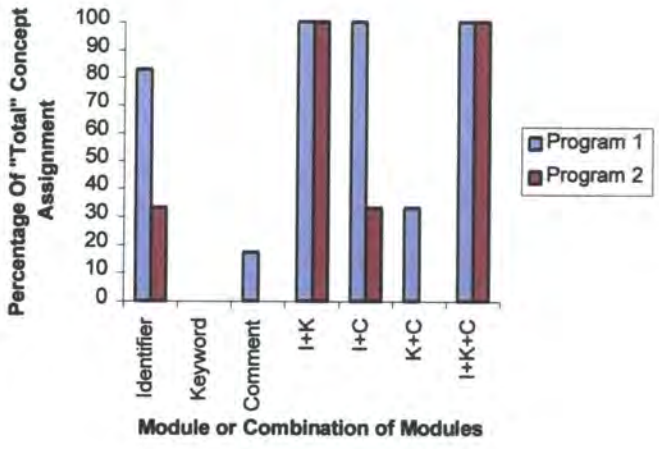


Figure 92: Chart to show the Proportion of "Total" Concept Assignment Achieved by Indicator Recognition Modules

Note that the "total" concept assignment is regarded as that achieved by all three modules in combination, shown in the rightmost bars on the chart (category I+K+C).

It is clear that the identifier and comment classes contribute the majority of the information and are capable of providing enough for concept assignment in the absence of any other modules. Although keywords do not appear to contribute enough hypotheses for concept assignment when considered alone, it is interesting to note that in both cases shown above, keywords in combination with other modules produce better assignment than those modules individually.

The performance of individual modules is dependent on the proportion of appropriate tokens in the source. Comments do not appear to contribute as much as might be expected but this can be attributed to a lack of comments in the short test programs. In all cases, identifiers seem the most useful source of information with comments taking second place. In the examples shown, keywords play a small but important role in augmenting the main sources of information.

The results of this investigation suggest that for the classes of indicator defined, the potential gains outweigh the relatively low cost of extraction. This may not be the case if, for example, program plans were used as indicators because the cost of searching for them can be extremely high. This would also apply to natural

language recognition of comment phrases rather than single words. The greater accuracy provided by such indicators may be beneficial but the relatively low cost of a plausible reasoning concept assignment system could be compromised.

It is not surprising that comments and identifiers contribute most of the information for concept assignment as they have greater domain semantics than keywords. Since the object of concept assignment is to label the source code with domain concepts, indicators with strong domain semantics will be more helpful than those without. Strong program semantics (such as those provided by keywords) are more helpful with structural information and consequently, are used best in areas such as segment boundary recognition.

It should be noted that, in addition to requiring indicator classes with strong domain semantics, such information also needs to be available within the source code. These are effectively two sides of the same problem. A maintainer's understanding of source code without meaningful identifiers or comments, is restricted largely to program-level semantics. Using solely the keyword indicator recognition module restricts HB-CA's view in a similar way. If source code with meaningless identifiers is analysed then indicator matching will be unsuccessful, or the results will be confusing. In this respect, HB-CA suffers similar confusion to a maintainer attempting to understand poorly written code.

The sequential list model employed for hypotheses has many strengths, in particular, the ability to integrate multiple knowledge sources as discussed earlier in this section. There are also some disadvantages to the approach, e.g. the problem of representing structure-based indicators. This is discussed in more depth in section 8.9.

8.9 Representational Power

This section discusses the ability of HB-CA to represent different types of indicator and concept.

The library was originally designed to represent indicators as textual tokens and would require significant modification to represent other types of information. It might be possible to use the tokens to refer to a file or other container capable of storing more complex indicators such as program plans. These require a representation capable of modelling data and control flow constraints. Plans such as compute-hash could be stored as a name but there would be difficulty in reliably recognising them without the complex indicators.

The problem of representing information about relationships and constraints is illustrated further by extending the airline-booking example suggested in [BIGG93]. Although the concept "reserve airline seat" could be modelled using diverse evidence for the constituent parts of the process, expressing a constraint such as "only one person can reserve one seat" is much more difficult. The domain-specific relationships between objects in the program cannot be modelled in the library as it only allows composition and specialisation. Consequently, there is no easy way of describing the evidence for the concept. The library's ability to model business rules is clearly affected by this, since the definition of a business rule (see Chapter 2) specifies that it is a requirement on the condition or manipulation of data. Rules can be modelled in terms of the features involved in the manipulation of the data such as likely variable names and computational keywords. This does not guarantee to find the business rule in its precise form but if the rule's implementation is coherent in the source code, the concept name describing it may be assigned correctly.

The assumption underlying the library's representation of concepts is that of spatial co-occurrence. If several pieces of evidence for a concept occur in close proximity to each other then that concept can be determined. It would not be impossible to assemble evidence for constraints in the library, but reliably achieving accurate concept assignment for them seems unlikely.

Much of the abstracting power of HB-CA's representation is derived from the ability to store any concept name desired by the user. Very high-level abstractions may not be found successfully if they correspond to larger sections of program than those to which the method aims to assign concepts.

In addition to the limited representation of indicators as simple tokens, HB-CA could be hampered by its ordered hypothesis list. This would cause particular problems for those indicators relying on spatial relationships between parts of the code, e.g. a delocalised program plan. Although the plan might be found within a section, it would be difficult to decide where to place its hypothesis in the list since it participates at several disjoint points in the code. One solution would be to create a hypothesis for every line of the plan but this may upset the balance of evidence in the appropriate segment. The problem would be less severe for natural language phrase indicators as they are likely to occur on a single line.

Another problem with the ordered hypothesis representation is that no account can be taken of an indicator's type or other syntactic properties. In performing the investigations for this chapter, it has been observed that concept assignments are made occasionally to sections of source code that have been commented out. This is because the comments have not been recognised as such by the segmentation or concept binding stages. The issue could be addressed by examining the type of the indicators within a segment and rejecting that segment if no executable code is found. Taking account of other syntactic properties could improve the general performance of HB-CA, e.g. if a particular token was known to be a section name, it could be given greater weight than the other indicators in the segment and provide a context for their examination. The risk with this approach is that more reliance is placed on a single token than on the general body of evidence. This risk was deemed unacceptable in HB-CA, leading to the "naïve" token model in use.

8.10 Domain Independence

Although HB-CA's algorithms are not tailored to a particular domain, its reliance on a domain model means that a library developed for one domain may not transfer to others without significantly impairing concept assignment performance. To investigate this, the library content presented in the Appendix was applied to several programs from a different system to that providing the other examples in this chapter. Whilst not from a significantly different domain, these programs serve to illustrate some of the issues associated with domain independence. The investigation parameters are shown in Table 20.

Program Set	Set H
Library Content	Appendix, Section A.2
rec_thresh	1
min_vd	3
forced_specialisation	True
Results	Appendix, Section A.8

Table 20: Parameters for Investigation of Domain Independence

Results from both domains are shown in Table 21 (see Table 13 for other results from the old domain). Programs from the new domain show markedly lower average accuracies.

	New Domain	Old Domain
Mean Accuracy	44%, σ = 29	84%, σ = 14
Mean Strict Accuracy	30%, σ = 26	56%, σ = 19
Median Accuracy	53%	89%
Median Strict Accuracy	26%	50%

Table 21: Average Accuracies for Library Applied to a Different Domain

The concepts found in the alternative system are more general than those from the original, and make less use of specialisations in the library.

8.11 Language Independence

HB-CA was designed to work with the COBOL II language.

The use of hypotheses as the primary reasoning component of HB-CA gives it the potential to be applied to other languages. The following sections discuss potential issues that may arise from such applications.

8.11.1 Imperative, Non Object-Oriented (e.g. C, Pascal)

This kind of language is similar to COBOL II and could be readily analysed by HB-CA. Assuming appropriate lexers are used for extracting tokens in the indicator classes, the matching stage of hypothesis generation would not require modification. Since the remainder of the method relies on hypotheses rather than source dependent information, no changes should be required to these stages either. The knowledge base would need to be equipped with indicators appropriate to the new language, particularly in the keyword class. Minor modifications could be made to exploit the scope of variables in block-structured languages, but in principle, this should not be a major issue.

8.11.2 Imperative, Object-Oriented (e.g. C++, Delphi, Java)

Applying HB-CA to object-oriented languages may not be as successful as applying it to those described in the previous category. HB-CA adopts a linear view of source code in a file, regarding the file as containing one program made up of a number of subroutines. Superficially, the class definition of an object-oriented language could be seen in a similar fashion, with methods regarded as subroutines. There are important differences, e.g. not all of the information required to assign a concept to a method may be in the file as much of it may be inherited from super-classes. Analysing this additional information would require substantial modification of HB-CA to enable it to handle a collection of files. In addition, the nature of object-oriented programming means that related functions and data structures tend to be grouped within a single class. Methods may be smaller than their equivalent procedures in another language (due to encapsulation and scoping), which may lead to easier comprehension. The functional grouping and smaller method size make it less likely that concept assignment would be of great benefit.

In summary, HB-CA could be applied to an object-oriented language with minor modification but the benefit of applying this type of tool may not be worth the effort required.

8.11.3 Non-Imperative (e.g. Haskell, Prolog)

These languages challenge many of the assumptions on which HB-CA rests, e.g. the notion of sequence between program statements, subroutines to provide basic segmentation, and the availability of a reasonably large body of evidence within the code to indicate functionality.

Programs written in functional languages such as Haskell do contain a certain amount of informal information, and concept assignment might be attempted. The generally limited size of such programs and the style of programming adopted make this an exercise of dubious merit.

There is a large amount of literature on the psychology of understanding programs written in imperative languages but very little on functional or logic languages. Some examples of work on the latter type are [ROME99], and [HAZA93]. The general lack of research in this area could be due to the lesser financial imperative of maintaining systems written in these languages. Since such work rightly forms the basis for the design of program understanding tools and methods, there is a need for more investigation to establish the requirements and feasibility of tool support for logic and functional languages.

8.12 Cognitive Requirements

This section evaluates HB-CA (as implemented in HB-CAS) against the cognitive design element framework described in [STOR97] and [STOR98]. The version used here is drawn from [STOR98] in which two of the elements from [STOR97] appear to have been combined. The framework (shown in Figure 93) is designed to guide the development and evaluation of software exploration and comprehension tools. Where possible, the criteria are discussed with reference to the HB-CA method but those that are clearly implementation-specific refer to HB-CAS.

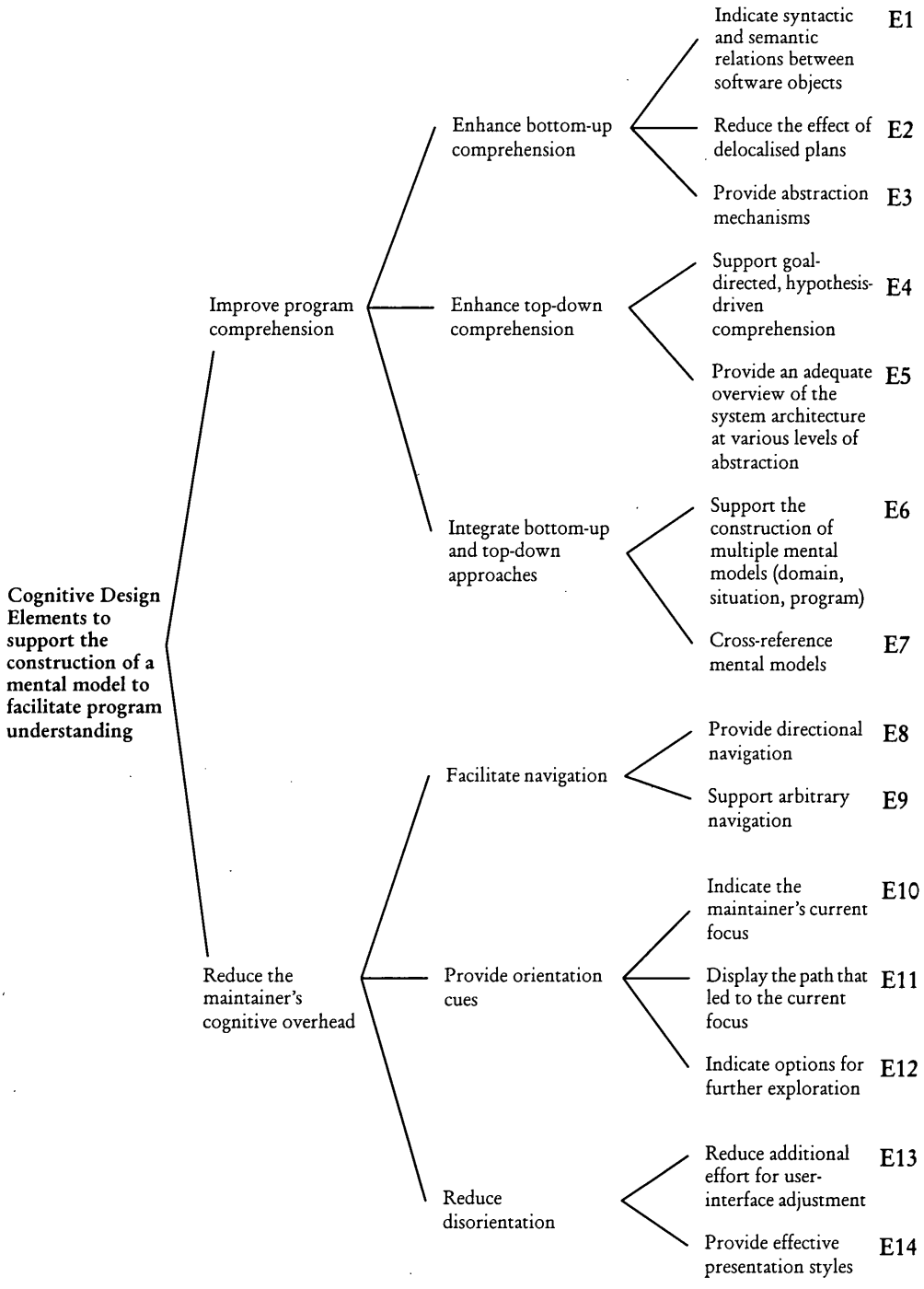


Figure 93: Cognitive Design Elements for Software Exploration Tools [STOR98]

There are fourteen elements divided into two main branches. The first aims to capture the essential processes of various comprehension strategies, and the second addresses the cognitive overhead experienced by a maintainer exploring software

[STOR98]. A description of the various comprehension strategies can be found in Chapter 2.

8.12.1 Improve Program Comprehension

8.12.1.1 Enhance Bottom-Up Comprehension

E1: Indicate syntactic and semantic relations between software objects.

- Immediate and visible access to low-level program units (such as source code) should be provided. The syntactic and semantic relations of these units should be clearly visible [STOR98].
- HB-CA only analyses one unit of source code at a time. Relationships between units are not supported.

E2: Reduce the effect of delocalised plans.

- A delocalised plan results from the fragmentation of source code related to a particular algorithm or plan. Understanding this can be disorienting or cumbersome without tool assistance [STOR98].
- HB-CA does not undertake plan analysis of a program. Some plan-type groups may be detected by the concept assignment methods but only through the informal evidence available. Delocalised plans are liable to be detected as either one large segment containing a high proportion of unrelated code, or as a series of smaller segments within the delocalised plan. HB-CAS has no facility for hiding unrelated parts within the plan.

E3: Provide abstraction mechanisms.

- Storey claims that maintainers may benefit from creating their own abstractions and labelling them to reflect their understanding. This might help them to better comprehend the software than if they use prefabricated views provided by a tool [STOR98].
- HB-CAS reflects the maintainer's current understanding of the domain (represented in the knowledge base) rather than a specific program. Naming of domain concepts can be performed in accordance with the maintainer's requirements. The purpose of HB-CA is to alleviate the effort of applying these to the source code.

8.12.1.2 Enhance Top-Down Comprehension

E4: Support goal-directed, hypothesis-driven comprehension.

- This requires the maintainer to possess prior application-domain knowledge, previous exposure to the program, or access to its documentation. Understanding is performed depth-first through hypothesis formulation and verification [STOR98].
- HB-CA supports hypothesis-driven comprehension in a limited way. If a maintainer formulates hypotheses about the functionality of a program, these can be swiftly verified with the concept list provided by HB-CAS. In addition, formulating the knowledge base will require exploration of domain knowledge by the maintainer.

E5: Provide an adequate overview of the system architecture at various levels of abstraction.

- HB-CA works on single modules of code and does not aim to support system-level analysis.

8.12.1.3 Integrate Bottom-Up and Top-Down Approaches

E6: Support the construction of multiple mental models (domain, situation, program).

- The variety of models that may be used by a maintainer during comprehension have been unified in a meta-model (see [MAYR95]). Ideally, software tools should support any model required by the maintainer through multiple views [STOR98].
- HB-CA supports multiple models by identifying concepts. These can be used in a bottom-up context for abstraction, or top-down for hypothesis verification. Although support for the situation and domain models is reasonably good, HB-CA does not assist greatly with building the program model.

E7: Cross-reference mental models.

- HB-CAS does not support cross-referencing between views of a system because of its single module approach.

8.12.2 Reduce the Maintainer's Cognitive Overhead

8.12.2.1 Facilitate Navigation

E8: Provide directional navigation.

- Directional navigation refers to reading source code and documentation sequentially, browsing the source code using data and control flow relationships, traversing software structure in hierarchical abstractions, and following user-defined program or application dependent links [STOR98].
- HB-CAS supports this by providing hypertext links between the concept list and source code browser.

E9: Support arbitrary navigation.

- Arbitrary navigation is supported when a maintainer navigates to locations not necessarily reachable by defined links [STOR98].
- Although HB-CAS provides hypertext linkage between concepts and display, it does not support this kind of arbitrary navigation.

8.12.2.2 Provide Orientation Cues

E10: Indicate the maintainer's current focus.

- This refers to the process of showing the maintainer's object of interest and its context. Textual views of source code implicitly show the focus, although related areas of code may not be visible [STOR98].
- HB-CAS supports a textual view but does not aim to provide contextual information of the type discussed above.

E11: Display the path that led to the current focus.

- Recording why a maintainer is interested in a particular object is very important [STOR98].
- HB-CA does not aim to capture this information.

E12: Indicate options for further exploration.

- This refers to the way in which a user is made aware of facilities for further exploration [STOR98].
- HB-CAS does not provide more than one way of exploring code.

8.12.2.3 Reduce Disorientation

E13: Reduce additional effort for user-interface adjustment.

- Poorly designed interfaces induce an additional overhead and available functionality should not impede the program understanding task [STOR98].
- HB-CAS was designed as a research prototype. Consequently, greater emphasis is placed on intermediate data structures and process monitoring than would be required for a real-world system. The source code browser clearly shows the results of the method although there is substantial scope to improve it.

E14: Provide effective presentation styles.

- In this criterion, Storey discusses graph layout almost exclusively. This has no relevance to HB-CA.

8.12.3 Summary

This section has used Storey's cognitive design element framework to evaluate HB-CA and HB-CAS. In most of the areas, either HB-CA or HB-CAS adequately meets the criteria specified, failing only those that are beyond the original scope and objectives of the work.

8.13 Summary

This chapter has presented the first part of an extensive evaluation of HB-CA. Beginning with an investigation of the scalability properties of the method, the discussion has covered research and design characteristics, highlighting the successes and failures of HB-CA's approach to concept assignment.

Chapter 9 contains the second part of the evaluation, looking at the applicability of HB-CA in the software maintenance process. Several applications are identified and their associated cost issues discussed.

Chapter 9
Evaluation II: Applications of HB-CA

9.1 Introduction

Chapter 8 presented an extensive evaluation of many characteristics of HB-CA. It discussed properties such as scalability, computational cost, and representational power. Strengths and weaknesses of the techniques employed for segmentation and concept binding were highlighted. Suggestions were made for solving some of the remaining problems in the HB-CA method.

This chapter concludes the evaluation by examining applications of HB-CA in the software maintenance process. It is shown to have potential benefit for several activities.

9.2 HB-CA in the Software Maintenance Process

The discussion of the software maintenance process in Chapter 2 was based on the IEEE standard [IEEE98] and identified several areas where software comprehension was required. These were parts of the analysis, design, and implementation stages of the standard process, and are now revisited to explore the potential benefit of HB-CA.

The activities can be divided broadly into two categories: those that assist with analysing a change, and those concerned with making the change. The first category consists of business-rule ripple analysis, code ripple analysis, and module selection. The second consists of software module comprehension only. The way in which HB-CA could help to reduce the cost of these activities is discussed in the next few sections, making the assumption that HB-CA is accurate, and that the library is complete with respect to the concepts required. Any cost savings achieved through the use of HB-CA would be offset by less than perfect accuracy or an incomplete library.

9.2.1 Analysis Activities

9.2.1.1 Business-Rule Ripple Analysis

Ripple analysis occurs in the analysis phase of the IEEE standard. Business-rule ripple analysis involves determining the potential effect of changing a business rule on other parts of the system (an example of higher-order impact analysis, see [TILL96b]). Using HB-CA to assist with ripple analysis would require some additions to the method or its implementation to enable it to analyse multiple source files. This would be a wrapper supplying each candidate file to HB-CA and analysing the result of concept assignment. Business rules would be modelled in the library, and the modified method would be supplied with the rule being proposed for change. Section 8.9 discusses issues relating to business-rule modelling. The library would need to be populated entirely with business rule concepts to avoid confusion with the programming domain. Concepts found in programs that implement the "proposed change" rule would be presented as candidates for side effects of the change. The maintainer could then examine them and accept or reject these suggestions for further analysis. The process is shown in Figure 94.

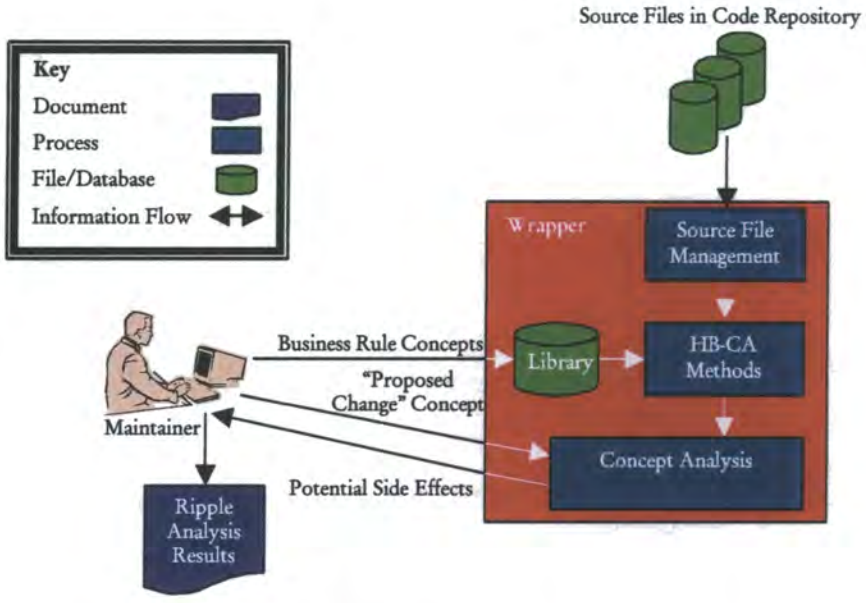


Figure 94: Diagram showing HB-CA used for Business-Rule Ripple Analysis

Chapter 2 stated that the cost of business-rule ripple analysis is crudely proportional to the number of artefacts inspected. One advantage of using HB-CA in this

activity would be that many programs could be scanned quickly for potential side effects in the business rules, reducing the number that the maintainer is required to examine by hand. This application of HB-CA may have limited success given the difficulty of representing constraint information in the library (see section 8.9). HB-CA could not totally replace the maintainer because it cannot determine dependencies between rules beyond that of co-occurrence in the same program. In this sense, it does not undertake traditional ripple analysis as it does not predict the effect of a change, but only makes suggestions for potential side effects. If it is likely that related rules do co-occur, HB-CA could substantially reduce the size of the task by limiting the number of code items that require inspection.

9.2.1.2 Code Ripple Analysis

This is similar to business-rule ripple analysis but is more likely to occur in the design phase of the maintenance standard as part of identifying affected software modules. Code ripple analysis is used to determine the effect of changes to the source code. There are various methods to perform this using syntactic and semantic techniques (e.g. forward program slicing, see [NING94]), but HB-CA could perform it on a conceptual level. There is little difference between code ripple analysis and business-rule ripple analysis, except in the type of concept being considered. Business rules are closer to the application domain than the type of concepts that usually would be used for code ripple analysis. These would probably be nearer to the implementation domain. The process of using HB-CA for this activity would be much the same as that shown in Figure 94, although the library would probably contain lower-level concepts in addition to those modelling business rules.

Chapter 2 stated that the cost of code ripple analysis is crudely proportional to the number and size of the artefacts examined. Potential cost savings could result from the reduced size of the code repository requiring manual inspection, on the principle that co-occurrence of concepts indicates some dependency. As discussed in section 9.2.1.1, relying solely on this relationship prevents HB-CA from fulfilling the requirements of traditional ripple analysis.

9.2.1.3 Module Selection

HB-CA can assist with this activity to a greater extent than it can with ripple analysis. Module selection can take place before and/or after ripple analysis, primarily occurring in the design phase of the standard process as part of identifying affected software modules. Once the concepts to be changed are known, the task of finding instances of them in the code base can be extremely time consuming. Using a similar wrapper to that described in section 9.2.1.1, the concept required can be supplied to HB-CA (as the only concept in the library) and programs that implement it can be found. These would be the modules requiring change.

Chapter 2 described the cost of module selection as a function of the size of the code repository and the search method. The cost savings from this application of HB-CA could be quite considerable since the maintainer does not need to participate in the selection activity if the wrapper is used. If HB-CA is employed in its current form (i.e. analysing one module at a time), reduced cost could still be achieved because the maintainer would not need to read every program entirely. The concept list would show whether the concept to be changed exists in the code. Concept-based search could perform better than some other automated methods of examining source code (e.g. plan recognition) because it has linear computational growth with the length of source code being analysed.

9.2.1.4 Code Reuse

Although not explicitly placed in the standard process, code reuse can substantially reduce the cost of software maintenance. Using HB-CA in a similar manner to module selection could facilitate this activity. It might be particularly helpful with languages such as COBOL II that do not lend themselves to populating reuse libraries. The code repository could be searched for instances of a particular concept required for implementation in another program. HB-CA could be particularly helpful since SOM-based segmentation may be able to identify parts of subroutines that implement the required concept, even if the whole routine is not relevant.

9.2.2 Implementation Activities

One of the steps in the implementation stage of the standard process is coding and unit-testing. Coding can be further subdivided into two stages: module comprehension, and change implementation (see [GALL91]). These may be iterative. Module (or program) comprehension is required in all of the above stages to some extent, but a greater depth of understanding is likely to be required for implementation. Recall that the comprehension activity is regarded as the translation of source code to another representation. In the case of concept assignment, the other representation is concept names labelling parts of the source code.

Figure 95 shows the module comprehension activity without HB-CA.

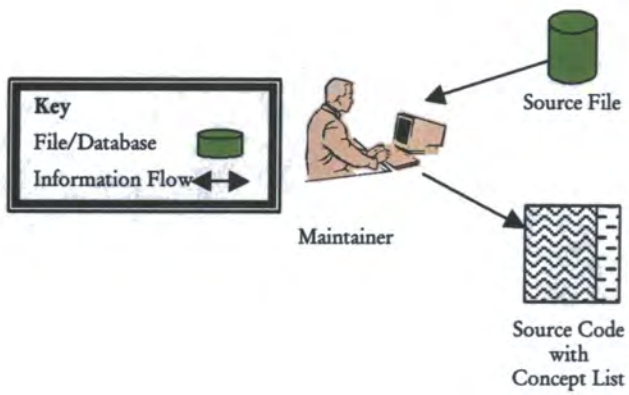


Figure 95: Module Comprehension without HB-CA

Figure 95 can be placed in the context of the comprehension activity framework for concept assignment described in Chapter 2 (Figure 7). This is shown in Figure 96.

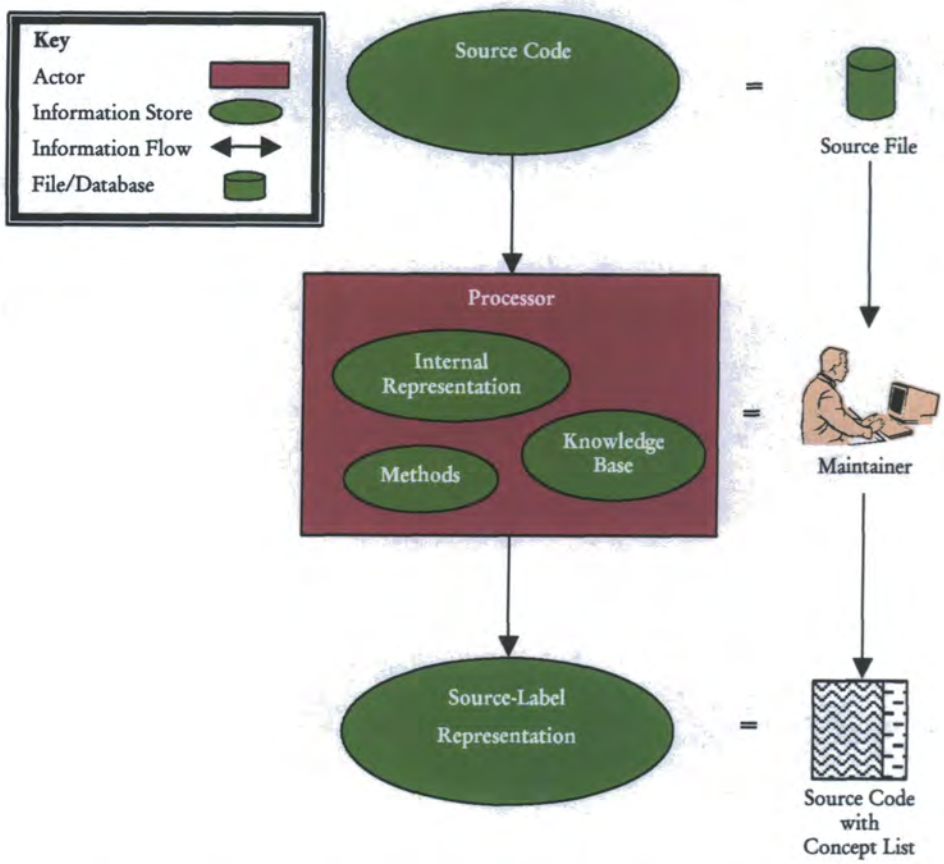


Figure 96: Module Comprehension Activity in the context of the Comprehension Activity Framework

The maintainer must undertake the work of summarising and abstracting the module to a mental or physical representation such as the concept list shown.

If HB-CA is used (see Figure 97) then much of the effort could be alleviated by providing the maintainer with the concept list automatically. Note that the library is shown as a file although strictly this is an implementation characteristic.

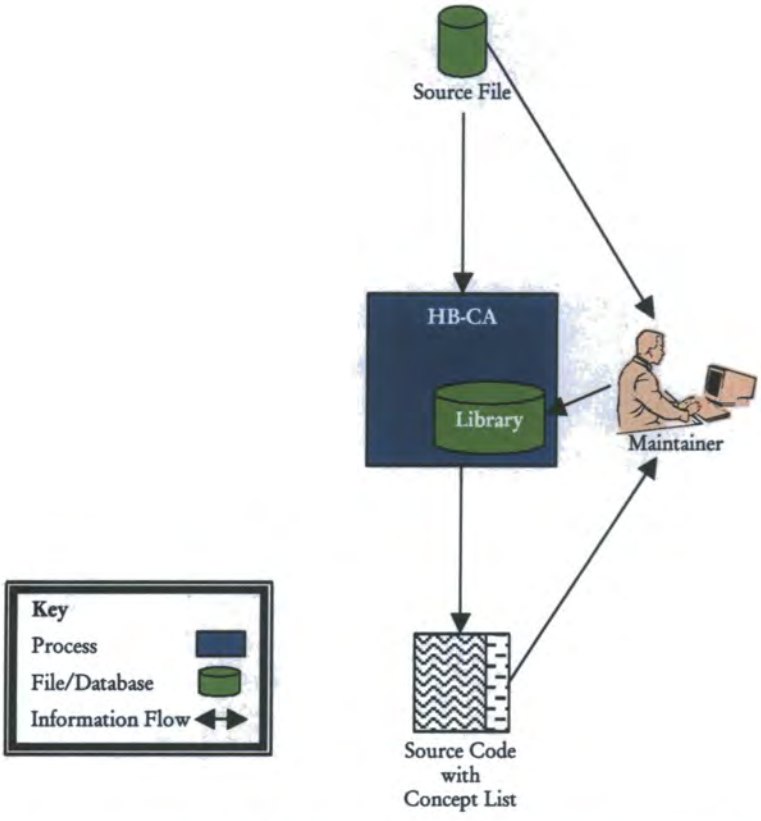


Figure 97: Module Comprehension with HB-CA

Note the feedback loop between the maintainer, library, and concept assignment process. By continually improving the domain model as HB-CA is used, the maintainer can increase the quality of results produced, thus further reducing comprehension time with every iteration. To compare the relative costs of the automated and manual approaches to concept assignment, the process in Figure 97 can be placed in the context of the comprehension activity framework. This is shown in Figure 98.

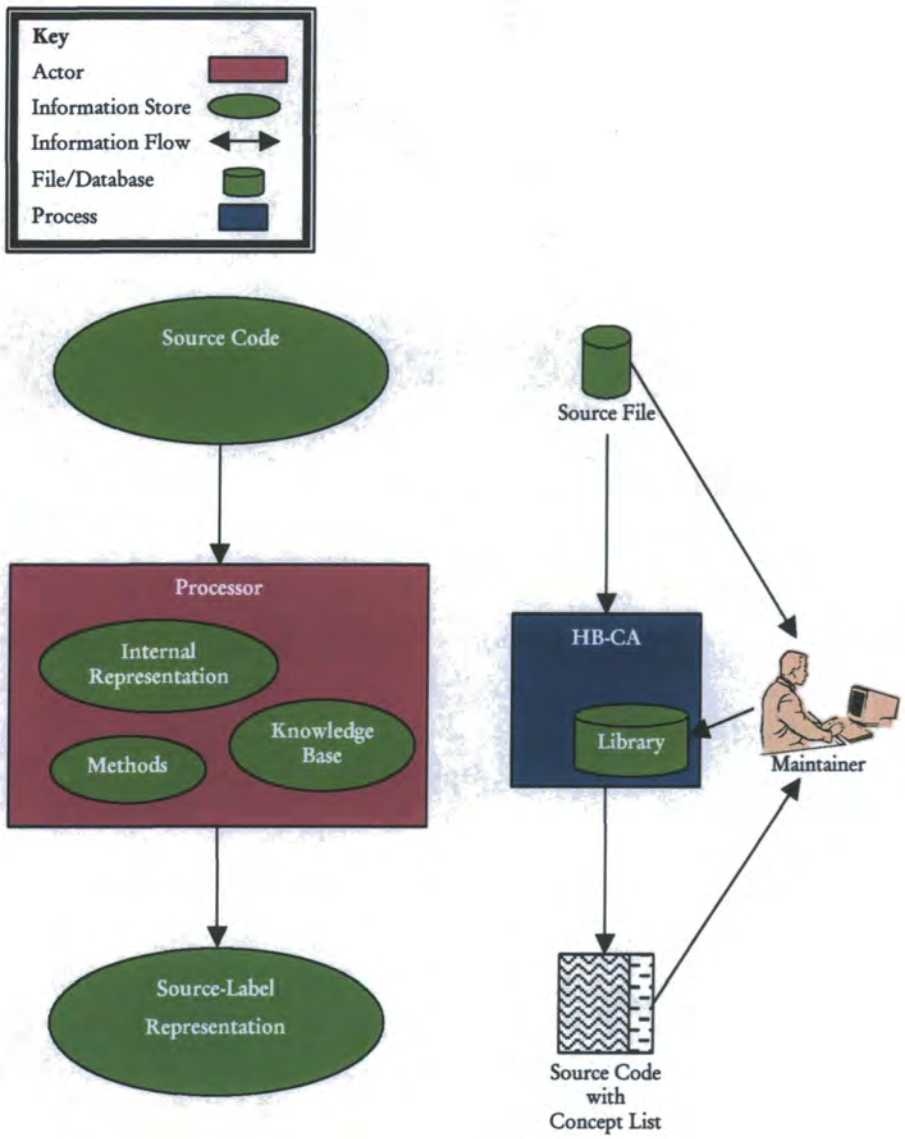


Figure 98: Module Comprehension Activity using HB-CA, in the context of the Comprehension Activity Framework

Using HB-CA saves the maintainer from performing the translation process, enabling them to begin their understanding using more than one knowledge source.

Since HB-CA fulfils the requirements of the comprehension activity framework, the relative costs of the automated and manual approaches can be compared. Performing concept assignment using automatic plausible reasoning should be less expensive than undertaking the task manually, and consequently using HB-CA should reduce the overall cost of software module comprehension. Providing automatic concept assignment also allows experienced maintainers to pass on their

domain understanding (via the domain model), helping less experienced maintainers to become familiar quickly with the system being maintained.

9.3 Summary

This chapter has discussed applications of HB-CA in the software maintenance process. These have been related to the activities identified in Chapter 2 as part of the IEEE software maintenance standard. Five maintenance activities could be assisted to varying degrees. Finding co-occurring concepts for further investigation might reduce the cost of business-rule ripple analysis and code ripple analysis. Module selection could be performed automatically with conceptual searching, and code reuse could be facilitated easily. HB-CA could assist in module comprehension by automatically providing a concept list related to the source code, thus relieving some of the comprehension burden from the maintainer. The reduction in cost when using HB-CA for software module comprehension has been discussed in the context of the comprehension activity framework defined in Chapter 2. If HB-CA reduces the cost of any of the activities described, it should achieve a reduction in the overall cost of software maintenance.

Chapter 10 concludes this thesis by summarising aspects of the concept assignment problem and the solution presented here. The success of HB-CA is discussed and ideas for further research are suggested.

Chapter 10
Conclusions

10.1 Introduction

Chapter 9 presented the second part of the evaluation, considering applications of HB-CA in the software maintenance process. Five areas were identified for potential cost reduction using HB-CA.

This chapter reviews the research presented in this thesis. The work accomplished is compared to the criteria for success defined in Chapter 1, some general issues are discussed, and directions for further work identified.

10.2 Review of Research

10.2.1 The Concept Assignment Problem

The concept assignment problem was defined in Chapter 1 as:

> "The process of assigning descriptive terms to their implementation in source code, the terms being nominated by a user and usually relating to computational intent."

Segmentation and concept binding were identified as the major research issues within this. Segmentation involves finding the location and extent of concepts, and concept binding determines which concepts are implemented at these locations.

10.2.2 Comprehension Activity Framework and Formal Model

Chapter 2 explored a number of aspects of software comprehension to create a framework that describes the activity in terms of translating one representation of software to another. This was based on factors common to psychological theories of program comprehension and common elements of software tools. The general framework was specialised for the concept assignment problem with specific source and target representations defined.

In conjunction with the specialised comprehension activity framework, a formal model (in set theory) was developed to define the representations more precisely. As each part of HB-CA was presented, its position in the framework was shown. The formal model was extended to capture the intermediate representations between stages. Chapter 6 summarised the formal model by drawing together all of the definitions in a single section.

10.2.3 Hypothesis-Based Concept Assignment

The solution to the concept assignment problem presented in this thesis is termed Hypothesis-Based Concept Assignment. It is a three-stage method addressing the two major research issues reviewed in section 10.2.1. A simple knowledge base is used to model pertinent aspects of the domain. This was described in Chapter 3.

HB-CA begins with hypothesis generation (presented in Chapter 4), comparing indicators stored in the knowledge base to tokens of various classes extracted from the source code under analysis. When an indicator matches a token, hypotheses are generated for every concept to which it is linked. The individual lists of hypotheses from the various classes are combined in order of occurrence and passed to the next part of the process: segmentation.

The segmentation stage (presented in Chapter 5) groups related hypotheses to form segments in the combined list. This is undertaken initially using hypotheses generated from subroutine boundaries, to ensure that the original program's structure is reflected in the resulting hypothesis segment list. Each segment is analysed further to determine whether enough hypotheses exist to potentially form two or more clusters within it. If this is the case, a self-organising map is employed to associate nearby, similar hypotheses. The resulting clusters are checked to ensure that sufficient evidence for concept binding is available within each. Any that have insufficient evidence are combined with neighbouring clusters that do, and are converted to segments.

The final stage of HB-CA (presented in Chapter 6) is concept binding. Each segment created in the previous stage is analysed to determine which concept has the most evidence. This is performed by generating initial conclusions from the

hypotheses in a segment and extending the conclusion list by exploiting knowledge-base relationships. The conclusions are scored using the available evidence and the highest scoring conclusion is declared the winner. In cases where more than one highest-scoring conclusion exists, a number of rules are applied to disambiguate the result. If these fail, the first conclusion is picked.

When all segments have been considered and their concepts bound, the process is complete and the source code can be labelled with the concept names.

The chapters describing the HB-CA method extended the formal model to capture data structures relevant to the stage of the process being presented. Each chapter compared HB-CA's approach to those adopted by two other plausible reasoning concept assignment systems: DM-TAO, and IRENE. A summary of this comparison is shown in Chapter 3.

10.2.4 Hypothesis-Based Concept Assignment System (HB-CAS)

In accordance with the research aims, the HB-CA method was embodied in a prototype software tool termed the Hypothesis-Based Concept Assignment System (HB-CAS). The architecture reflects the stages of HB-CA, employing separate programs to implement various parts of the process. The system was described and evaluated briefly in Chapter 7.

10.2.5 Evaluation

Chapters 8 and 9 presented an extensive and detailed evaluation of the HB-CA method using the criteria outlined in Chapter 1. The first part of the evaluation (contained in Chapter 8) dealt with characteristics and properties of the method, beginning with its scalability. Despite theoretical expectations of accuracy at all lengths of source code, practical investigations undertaken with HB-CAS indicated that lower accuracy occurred with larger programs. This was attributed to the algorithms that reallocate hypotheses from invalid clusters. Further issues arising from this investigation were discussed with particular reference to segmentation, concept binding, and library content. Various aspects of the computational and spatial cost were examined and HB-CA was found to have a linear computational growth in the length of the source code being analysed. Individual indicator classes

have different effects on concept assignment performance and these were discussed in the context of HB-CA's ability to use multiple information sources. Representational power and domain independence also were examined. HB-CA is intended to operate on COBOL II but could be applied to other languages. This issue was discussed with reference to several different classes of programming language. The first part of the evaluation was concluded by considering to what extent HB-CA and HB-CAS fulfil cognitive requirements for program understanding tools.

The second part of the evaluation in Chapter 9 discussed various possibilities for using HB-CA in the software maintenance process, and five applications were identified. HB-CA could potentially reduce the cost of business-rule ripple analysis and code ripple analysis, although both these cases require an assumption that co-occurrence of concepts signifies dependency. Module selection and code reuse could derive greater benefit from the use of HB-CA. Finally, the potential was shown for a reduction in the cost of software module comprehension and Chapter 9 described the way in which HB-CA could be used for this purpose. The relative costs of concept assignment using automatic and manual approaches were discussed with reference to the comprehension activity framework.

10.3 Evaluation of Research

An evaluation of the research reported in this thesis is now presented in the context of the criteria for success and research aims given in Chapter 1. These are repeated here with a discussion of each.

1) *The definition of a framework for the activity of software comprehension. This should capture the essential processes and data structures involved in software comprehension, regardless of whether the actor (i.e. the entity undertaking the comprehension activity) is a person or a software tool.*

Chapter 2 defines a comprehension activity framework based on elements common to software tools and psychological theories. The framework expresses the comprehension activity as a process of translation from one software representation to another by means of a processor (which could be a person or software tool).

The framework is specialised to the representations required for the concept assignment problem.

2) *The creation of a formal model of the comprehension activity framework discussed in criterion 1 to define clearly its data structures.*

Definition of this model commences in Chapter 2, specifying the source and target representations of the concept assignment problem. The comprehension activity is expressed as a function mapping one representation to another.

3) *The development of a new method to undertake automatic concept assignment using a simple knowledge base. It should be capable of analysing real-world COBOL II code and successfully cope with poorly structured and monolithic programs, in addition to well-structured examples. The method should provide a software maintainer with automatically recognised concepts linked to regions of source code.*

A new method, termed Hypothesis-Based Concept Assignment, has been developed to perform concept assignment automatically on COBOL II programs. The HB-CA method is presented in Chapters 3 to 6. It has been evaluated using real-world code and achieves high recognition accuracy, although performance can fall when larger programs are analysed. This problem has been investigated and the cause linked to specific naïve algorithms within the method. HB-CA can cope successfully with poor structure in programs, using a self-organising map to establish regions of conceptual focus when structural information is insufficient.

4) *As part of criterion 3, the development of novel approaches to address the two main research issues in concept assignment: segmentation, and concept binding.*

Chapters 5 and 6 describe the methods used to address segmentation and concept binding. The application of SOMs to the segmentation task provides HB-CA with its ability to cope with monolithic and unstructured code, basing decisions about segments on the conceptual structure of the program rather than its syntax. The concept binding method assesses concept evidence using a combination of semantic network activation and disambiguation rules.

5) *The extension of the general formal model (see criterion 2) to the new concept assignment method.*

Throughout Chapters 3 to 6, the formal model is extended to capture the individual parts of HB-CA. The structure of the knowledge base is formally described and all intermediate representations are defined. Chapter 6 summarises the model by collating all the definitions. The model adequately describes all that is required of it and, although some definitions could be augmented, there would be little benefit from the exercise.

6) *The implementation of a prototype tool to demonstrate the feasibility of the new concept assignment solution. This should allow easy evaluation of the method.*

Chapter 7 describes and evaluates the HB-CAS implementation of HB-CA. The prototype successfully undertakes concept assignment on COBOL II source code and was used for the practical parts of the evaluation presented in Chapters 8 and 9.

It has been demonstrated that the work presented in this thesis meets the criteria for success and research aims defined in Chapter 1. Section 10.4 discusses these accomplishments, and section 10.5 identifies areas for continuing the work and improving the capabilities of the method.

10.4 Discussion

A reflective discussion of the work accomplished in this thesis is now presented.

In general, HB-CA is a success. It meets the requirements shown in Chapter 1 and has exceeded expectations in its recognition accuracy.

Synonym matching was slightly disappointing as it caused significant difficulties in implementation and showed poor performance. If it were to be included in other concept assignment systems, pilot studies would need to be undertaken to determine the cost-benefit of the idea. In the absence of synonym matching, indicator recognition has been very successful, demonstrating the value of meaningful identifiers and comments. The potential for confusion when comments

are not relevant to the code with which they are associated has not proved to be a problem with the examples tested.

One of the major successes of HB-CA has been the SOM-based segmentation algorithm. Although the reallocation methods have been identified for further work, the success of the SOM method has vindicated the underlying principle of creating a conceptual map of a program. The idea of a conceptual map formed the basis for solving the difficult problem of determining conceptual segmentation without performing concept binding first. Various attempts were made to map the "conceptual landscape" of a program and this was achieved with relative ease. Creating a decision rule to determine which "peaks" were valid and which were not, proved more difficult to attain. The SOM emerged as a fine-grained approach to associating similar concepts whilst allowing a simple vector density criterion to be used for decisions. It has proved to be a successful technique and some developments are suggested in section 10.5. The reallocation algorithms require additional work but more success may be achieved by eliminating the principle that every hypothesis should be preserved. This was originally included to ensure that enough evidence was available for concept binding. Experience has shown that in many cases there would be enough hypotheses to make bindings, even if invalid clusters were ignored.

Given the success of the SOM technique, and the ability to label output nodes with the concept that triggers them most frequently, it is interesting to consider whether the entire concept assignment problem might be translated to the SOM. Early work with SOMs in HB-CA attempted this task with very limited success but these experiences should not rule out further efforts in this direction. Modifications to HB-CA would be required because no intelligent exploitation can be made of the relationships in the knowledge base. Once hypotheses are passed to the map, they must all compete. The solution to this may entail the generation of composite and specialised hypotheses, as these would need to compete with the single forms of concepts. Creating a map for the entire program also entails considering the characteristics of the input space; syntactic boundaries would need to be encoded as discontinuities in the sequence, to prevent the cross-subroutine associations observed in early efforts.

Concept binding based on semantic network activation has proved to be a good idea. This is not particularly surprising since it can be seen as a coarse-grained connectionist approach with some similarity to that employed by DM-TAO. The algorithmic version encodes the more holistic view of the semantic-network scoring approach that was devised first. The disambiguation rules were derived by considering the principles and goals of HB-CA and they exhibit the desired characteristic of graceful performance degradation with conflicting hypotheses. There is ample scope to refine and improve them, particularly those that manage ambiguity when forcing specialisation.

The knowledge base has proved effective despite its simplicity. Most of the concepts used have been of moderately low levels of abstraction and it would be interesting to investigate further ways of encoding higher-level and business-rule concepts. One significant issue with the knowledge base is its inability to natively encode constraints. This was discussed briefly in section 8.9. It would be possible to overcome this limitation with the file-based indicator approach outlined in section 8.9, in combination with an indicator recognition module capable of detecting the type of constraint required. This type of concept could be arguably beyond the scope of concept assignment systems since it does not express computational intent, but computational restriction. Nonetheless, the information provided by such constraints is very useful in software comprehension. The danger of using any type of complex indicator is that the cost advantage obtained when using plausible reasoning concept assignment techniques (such as HB-CA) might be negated.

Comparing HB-CA to the systems DM-TAO and IRENE has proved an interesting exercise. HB-CA is similar to DM-TAO in the type of concept it seeks and the way in which it performs concept binding. However, it shares some features with IRENE such as a reasonably simple knowledge base and the ability to explain concept assignment decisions. It is unique in clearly separating the stages of segmentation and concept binding. The simpler knowledge base used by HB-CA offers potentially easier domain modelling than either of the other systems, but lacks the dependency modelling ability of IRENE and the wide variety of concept types

of DM-TAO. Despite these drawbacks, concept assignment can be performed successfully.

Overall, HB-CA has proved to be a successful concept assignment solution and has demonstrated the potential for conceptual mapping of programs.

10.5 Further Work

The work presented in this thesis could be extended in many ways and some ideas are discussed in this section.

10.5.1 SOM-Based Concept Assignment

Section 10.4 suggested the possibility of using the SOM to perform both segmentation and concept assignment functions. This would be an interesting variation on the existing method and might reduce its cost. Concept assignment could be performed by labelling each output node in the map with the name of the concept that triggers it most often. Vector density measures would still be required to provide a recognition threshold but other concept binding parameters should be unnecessary. There are implications for hypothesis generation in that every plausible hypothesis (composite and specialised) for an indicator would need to be generated. This would be similar to conclusion generation but with the whole program regarded as a single segment. Under these conditions, sensible conclusion generation would be difficult to achieve. A method would be required to resolve the tension between the need for a predefined segment for conclusion generation, and the attempt to execute both segmentation and concept binding in one step. These issues would be subjects for research.

10.5.2 Intelligent Reallocation Algorithms

The problems with naïve reallocation of hypotheses were highlighted in Chapter 8. Two approaches now are suggested to solve this problem. The first is to ignore any invalid clusters. The risk associated with this method is that occasional mis-association on the SOM could result in valuable information being lost. The alternative approach is to improve the way in which hypotheses are reallocated, by using their conceptual content as a guide rather than simply dividing clusters equally into their surroundings. Various heuristics could be derived to implement this, e.g.

a hypothesis could be compared to its nearest valid cluster; if the hypothesis does not already appear in the cluster then it should not be attached. Such heuristics may require experimental investigation to determine their effectiveness. Other non-conceptual characteristics could be used, e.g. the distance in lines between the indicators for the two hypotheses could guide the selection of an appropriate cluster. These changes could improve the quality of segmentation.

10.5.3 Richer Knowledge Base

Although one of the aims of this work was to perform concept assignment with a simple knowledge base, increasing its complexity could be a fruitful line of research. Incorporating more inter-concept relations (e.g. secondary hypotheses or multiple composites) could increase the representational power and concept assignment abilities of the method. It is important to realise that such changes may cause the creation and maintenance costs of the library to rise and any potential benefit should be weighed against this.

10.5.4 Richer Conceptual Map

The idea of a conceptual map was discussed in section 10.4. It would be interesting to extend this notion to build a more informative map by using syntactic and semantic characteristics of the source code. This map could be used as the basis for a visualisation system, or to improve the concept assignment ability of HB-CA. Knowing that a particular identifier is a subroutine name could provide a context for the evaluation of other information within the subroutine. The risks of applying this type of maxim are noted in section 8.9. Placing greater weight on the information provided by a particular identifier could improve the accuracy of concept assignment. However, if the identifier is misleading then there is a greater chance of incorrect assignment than when a uniform weighting model is used. The depth of information could be increased by using a "level of confidence" measure of the accuracy of indicator matching. Those indicators matched using sub-strings or synonyms would gain a lower confidence level than those matched directly.

10.5.5 Use of the Data Division

The current form of HB-CA is concerned solely with the procedure division of COBOL II programs. The data division contains much useful information and

could augment HB-CA's object concept acquisition. The identity of data structures could be determined by rigorous analysis, or through HB-CA's concept assignment routines. Either could help to reduce the number of possible objects for later consideration. HB-CA would require modification because it relies on subroutine boundaries for its segmentation and has no awareness of the structure of data declarations. Preliminary research in this area should establish whether the potential benefits in accuracy outweigh the effort of additional analysis.

10.5.6 Large-Scale Evaluation

The evaluation in Chapter 8 provides much useful information about the nature of HB-CA. Section 8.12 demonstrated that HB-CA and HB-CAS fulfil many of the cognitive requirements for program understanding tools. These properties could be investigated further by undertaking a large-scale study to determine the effectiveness of the tool when used in real maintenance situations. This could guide the development of further research on the method and tool, in addition to providing information about the effectiveness of this type of comprehension assistance.

Other forms of large-scale evaluation could involve testing HB-CA with considerably more complex library content, and larger source programs from different domains.

10.5.7 Software Evolution Study

Although HB-CA was intended as a software maintenance support method, it could be used as a research tool in its own right. It would be interesting to examine many versions of the same program and to study changes in concept assignment through the program's maintenance history. This may provide insight into the way concepts break down and move within the program, leading to more effective strategies for maintenance. Research in this area may need to establish the viability of concept assignment as a measure of comprehensibility before undertaking the study itself.

10.6 Final Summary

A review of the work accomplished has been presented in this chapter. The overall success of the research has been considered in terms of the criteria shown in Chapter 1, and several directions for further work have been established.

This thesis has examined the context, motivation, and definition of concept assignment, leading to the development of a framework to describe software comprehension, and a formal model of important representations. A new, automated solution to the concept assignment problem has been presented: Hypothesis-Based Concept Assignment. The stages of HB-CA have been described and compared to similar systems. An extensive evaluation has demonstrated various characteristics of the method including linear computational growth in the length of program being analysed, high accuracy, and the ability to operate on real-world programs of varying quality. The potential for HB-CA to be applied in several parts of the software maintenance process has been shown, and possible cost savings have been identified. Ideas for further work have been suggested.

Hypothesis-Based Concept Assignment is a novel and successful solution to the concept assignment problem.

Appendix
Investigation Data

A.1 Introduction

The appendix contains source data for the graphs shown in the evaluation, program sets, and the library content referred to by some investigations. Library content for the remaining investigations can be found in Chapter 3.

Most results are rounded to the nearest integer. High-resolution timings are given to two decimal places and low-resolution timings are truncated to the nearest integer. The truncation is performed internally in Delphi and is beyond the control of the programmer. Program lengths are given in lines including white space and comments.

A.2 Library Content Used in Sections 8.2, 8.4, 8.8, 8.10

```
Library Output: EvalObs2
==============================

Primary Action Concept: Output
------------------------
Indicators: KDisplay KEndWrite KWrite KIO KOutput COutput
Composites: File Report Database Record
Specialisations: NONE

Primary Action Concept: Read
------------------------
Indicators: NRead KRead CRead
Composites: File Database Record
Specialisations: NONE

Primary Action Concept: Write
------------------------
Indicators: KEndWrite KWrite NWrite CWrite
Composites: File Database Record
Specialisations: NONE

Primary Object Concept: File
------------------------
Indicators: NFile KFile KFileControl CFile
Composites: NONE
Specialisations: APSMasterFile CAF PaymentFile
```

```
Primary Object Concept: Report
------------------------
Indicators: NReport KReport KReporting KReports CReport
Composites: NONE
Specialisations: NONE

Primary Object Concept: Database
------------------------
Indicators: NDatabase NDB CDatabase CDB
Composites: NONE
Specialisations: CMS

Primary Object Concept: Record
------------------------
Indicators: NRecord KRecord KRecords CRecord
Composites: NONE
Specialisations: APSRecord

Primary Action Concept: Call
------------------------
Indicators: NCall KCall CCall
Composites: DATEPRESModule
Specialisations: NONE

Primary Action Concept: Update
------------------------
Indicators: NUpdate CUpdate
Composites: File Database Record Policy
Specialisations: NONE

Primary Action Concept: Input
------------------------
Indicators: KIO NInput KInput CInput
Composites: File Database Record
Specialisations: NONE

Primary Object Concept: DATEPRESModule
------------------------
Indicators: NDatePres CDatePres
Composites: NONE
Specialisations: NONE

Primary Object Concept: Policy
------------------------
Indicators: NPolicy CPolicy
Composites: NONE
Specialisations: NONE

Secondary Object Concept: APSRecord
------------------------
Indicators: NRecord KRecord CRecord NAPS NA.P.S CAPS
Composites: NONE
Specialisations: NONE

Primary Object Concept: Interest
------------------------
Indicators: NInterest CInterest
Composites: NONE
Specialisations: NONE

Secondary Object Concept: APSMasterFile
------------------------
Indicators: NAPS NA.P.S CAPS NMaster CMaster
Composites: NONE
Specialisations: NONE

Secondary Object Concept: CMS
------------------------
Indicators: NCIF CCIF
Composites: NONE
Specialisations: NONE

Primary Action Concept: Initialisation
------------------------
Indicators: NInitialisation CInitialisation
Composites: NONE
Specialisations: NONE
```

```
Primary Action Concept: Print
-----------------------
Indicators: NPrint CPrint
Composites: Report Record Cheque Heading
Specialisations: NONE

Secondary Object Concept: CAF
-----------------------
Indicators: CFile NCAF NC.A.F CCAF CCentral CAnnuity
Composites: NONE
Specialisations: NONE

Primary Object Concept: Cheque
-----------------------
Indicators: NCheque CCheque
Composites: NONE
Specialisations: NONE

Primary Object Concept: Heading
-----------------------
Indicators: NHead CHead NHeading CHeading
Composites: NONE
Specialisations: NONE

Primary Action Concept: Calculate
-----------------------
Indicators:  KAdd  KCompute  KDivide  KEndCompute  KEndDivide  KEndMultiply
KEndSubstract KEndAdd KGiving KMultiply KPlus KSubtract CCalculate
Composites: Interest
Specialisations: NONE

Secondary Object Concept: PaymentFile
-----------------------
Indicators: NFile KFile CFile NPayment CPayment
Composites: NONE
Specialisations: NONE
```

A.3 Data for Section 8.2: Scalability

A.3.1 Data for Figure 51, Figure 52, Figure 53, and Figure 54

Investigation Parameters: Table 10

A.3.1.1 *forced_specialisation* = True

Program	Length (lines)	Total Concepts	Accurate Concepts	Strictly Accurate Concepts	Number of SOMs Used	Percentage Accuracy	Percentage Strict Accuracy
gd95	89	3	3	2	0	100	67
gb92/6	190	1	1	0	1	100	0
gd25	238	6	6	4	0	100	67
gd12	285	8	7	4	2	88	50
gd30	337	6	5	2	2	83	33
gd60	387	10	10	7	2	100	70
gd91	441	9	8	6	3	89	67
gd96	491	12	12	11	1	100	92
gd83	547	15	11	7	5	73	47
gb64	596	14	13	7	3	93	50
gd26	650	14	7	6	3	50	43
gd81	701	6	4	3	2	67	50
gb73	728	21	20	12	4	95	57
gd28	807	16	15	7	4	94	44
gd67	879	15	11	8	5	73	53
gb01	1013	13	8	3	4	62	23
gd82	1105	26	16	8	7	62	31
gd02	1117	7	6	3	1	86	43
gb07	1162	35	28	19	6	80	54
gb03	1237	35	25	15	9	71	43
gbcm0133	1310	42	40	35	6	95	83
gb08	1374	37	33	21	5	89	57

Median Accuracy	89	Mean Accuracy	84	Standard Deviation (σ)	14
Median Strict Accuracy	50	Mean Strict Accuracy	56	Standard Deviation (σ)	19

A.3.1.2 *forced_specialisation* = False

Program	Length (lines)	Total Concepts	Accurate Concepts	Strictly Accurate Concepts	Number of SOMs Used	Percentage Accuracy	Percentage Strict Accuracy
gd95	89	3	3	2	0	100	67
gb92/6	190	1	1	0	1	100	0
gd25	238	6	6	4	0	100	67
gd12	285	8	7	4	2	88	50
gd30	337	6	5	4	2	83	67
gd60	387	10	10	7	2	100	70
gd91	441	9	8	7	3	89	78
gd96	491	12	12	12	1	100	100
gd83	547	15	15	10	5	100	67
gb64	596	14	13	7	3	93	50
gd26	650	14	11	7	3	79	50
gd81	701	6	4	3	2	67	50
gb73	728	21	20	14	4	95	67
gd28	807	16	15	7	4	94	44
gd67	879	15	13	9	5	87	60
gb01	1013	13	9	3	4	69	23
gd82	1105	26	18	8	7	69	31
gd02	1117	7	6	3	1	86	43
gb07	1162	35	26	19	6	74	54
gb03	1237	35	30	19	9	86	54
gbcm0133	1310	42	40	35	6	95	83
gb08	1374	37	33	21	5	89	57

Median Accuracy	89	Mean Accuracy	88	Standard Deviation (σ)	11
Median Strict Accuracy	56	Mean Strict Accuracy	56	Standard Deviation (σ)	21

A.3.2 Data for Figure 55

Investigation Parameters: Table 11

Segment Size (lines)	Total Segments	Accurate Segments	Strictly Accurate Segments	Percentage Accuracy	Percentage Strict Accuracy
0-10	39	35	26	90	67
11-20	28	25	14	89	50
21-30	15	10	2	67	13
31-40	3	3	0	100	0
41-50	8	8	1	100	13
51-60	2	2	0	100	0
61-70	2	1	0	50	0
70+	8	7	2	88	25

A.3.3 Data for Figure 56

Investigation Parameters: Table 11

SOMs Used	0	1	2	4	5
Mean Segment Size (lines)	12	70	11	4	5

A.3.4 Data for Figure 57

Investigation Parameters: Table 12

Program	Section Number	Total Concepts	Accurate Concepts	Strictly Accurate Concepts	Valid Clusters	Invalid Clusters	Total Clusters	Percentage Accuracy	Percentage Strict Accuracy	Percentage Invalid Clusters
gb73	1	4	4	1	4	2	6	100	25	33
	2	5	4	2	5	1	6	80	40	17
gd82	1	5	5	2	5	1	6	100	40	17
	2	5	2	0	6	2	8	40	0	25
	3	8	5	2	8	6	14	63	25	43
	5	3	1	0	3	1	4	33	0	25
gb03	1	2	1	0	2	1	3	50	0	33
	2	2	1	0	2	1	3	50	0	33
	4	12	9	4	12	2	14	75	33	14
	7	4	4	2	4	0	4	100	50	0
	12	3	3	3	3	0	3	100	100	0
	15	3	3	2	3	2	5	100	67	40
gd67	2	3	1	1	3	2	5	33	33	40
	3	3	3	1	3	1	4	100	33	25
	4	4	4	0	4	7	11	100	0	64
gd96	1	8	8	7	8	4	12	100	88	33
gd26	1	6	2	1	6	2	8	33	17	25
	3	4	3	3	4	2	6	75	75	33
gd30	2	3	2	0	3	1	4	67	0	25

A.4 Data for Section 8.4: Concept Binding

A.4.1 Data for Figure 74

Investigation Parameters: Table 14

Total Cases	101	
Rule	Triggered Instances	Percentage Triggered Instances
DAR 1	101	100
DAR 2	101	100
DAR 3	18	18
DAR 4	16	16
DAR 5	16	16
Arbitrary	10	10

A.5 Data for Section 8.6: Computational Cost

A.5.1 Data for Figure 75, Figure 76, and Figure 87

Investigation Parameters: Table 15

IRM 1: Identifier
IRM 2: Keyword
IRM 3: Comment
IRM 4: Segment Boundary

Program	Length (lines)	Number of Segments	Low Resolution Hypothesis Generation (IRM) Time (s)	Low Resolution Segmentation Time (s)	Low Resolution Concept Binding Time (s)	Low Resolution Total Execution Time (s)
gd95	89	1	3	0	1	4
gb92/6	190	1	9	0	0	10
gd25	238	5	12	0	3	16
gd12	285	3	17	3	3	24
gd30	337	1	22	0	1	23
gd60	387	4	25	0	2	28
gd91	441	4	29	0	6	36
gd96	491	10	36	4	10	51
gd83	547	10	37	4	7	48
gb64	596	7	38	1	7	47
gd26	650	9	40	4	10	54
gd81	701	4	49	0	3	52
gb73	728	11	44	7	11	63
gd28	807	6	61	4	4	69
gd67	879	9	70	10	5	86
gb01	1013	6	79	0	3	83
gd82	1105	10	92	10	10	113
gb07	1162	8	95	4	11	111
gb03	1237	17	100	5	18	124
gb08	1374	10	111	4	11	126

A.5.2 Data for Figure 77, Figure 78, and Figure 79

Investigation Parameters: Table 15

Program	Length (lines)	Low Resolution Execution Time (s)					Number of Tokens				
		IRM 1	IRM 2	IRM 3	IRM 4	Total	IRM 1	IRM 2	IRM 3	IRM 4	Total
gd95	89	1	1	1	0	3	32	71	59	11	173
gb92/6	190	1	3	5	0	9	43	128	189	5	365
gd25	238	3	6	3	0	12	105	233	117	15	470
gd12	285	4	10	3	0	17	127	398	125	11	661
gd30	337	5	7	10	0	22	165	290	359	11	825
gd60	387	8	15	2	0	25	258	580	72	19	929
gd91	441	9	19	1	0	29	252	721	63	13	1049
gd96	491	16	17	3	0	36	473	634	101	23	1231
gd83	547	12	19	6	0	37	369	720	207	25	1321
gb64	596	10	17	11	0	38	289	634	399	31	1353
gd26	650	13	19	8	0	40	373	704	298	22	1397
gd81	701	17	24	8	0	49	493	862	289	23	1667
gb73	728	12	18	14	0	44	352	686	477	30	1545
gd28	807	18	32	11	0	61	519	1144	379	17	2059
gd67	879	22	39	9	0	70	611	1358	306	23	2298
gb01	1013	24	36	19	0	79	704	1257	627	23	2611
gd82	1105	29	41	22	0	92	832	1395	720	27	2974
gb07	1162	32	48	15	0	95	892	1592	522	21	3027
gb03	1237	30	50	20	0	100	845	1654	669	33	3201
gb08	1374	26	37	48	0	111	739	1287	1444	35	3505

A.5.3 Data for Figure 80

Investigation Parameters: Table 15

	Percentage of Total Low Resolution IRM Time				Percentage of Total Extracted Tokens			
Program	IRM 1	IRM 2	IRM 3	IRM 4	IRM 1	IRM 2	IRM 3	IRM 4
gd95	33	33	33	0	18	41	34	6
gb92/6	11	33	56	0	12	35	52	1
gd25	25	50	25	0	22	50	25	3
gd12	24	59	18	0	19	60	19	2
gd30	23	32	45	0	20	35	44	1
gd60	32	60	8	0	28	62	8	2
gd91	31	66	3	0	24	69	6	1
gd96	44	47	8	0	38	52	8	2
gd83	32	51	16	0	28	55	16	2
gb64	26	45	29	0	21	47	29	2
gd26	33	48	20	0	27	50	21	2
gd81	35	49	16	0	30	52	17	1
gb73	27	41	32	0	23	44	31	2
gd28	30	52	18	0	25	56	18	1
gd67	31	56	13	0	27	59	13	1
gb01	30	46	24	0	27	48	24	1
gd82	32	45	24	0	28	47	24	1
gb07	34	51	16	0	29	53	17	1
gb03	30	50	20	0	26	52	21	1
Gb08	23	33	43	0	21	37	41	1

A.5.4 Data for Figure 81, Figure 82, Figure 83, and Figure 84

Investigation Parameters: Table 15

Program	Number of Sections	Number of SOMs Used	Low Resolution Segmentation Time (s)	High Resolution Segmentation Time (s)
gd95	5	0	0	0.16
gb92/6	2	0	0	0.06
gd25	7	0	0	0.44
gd12	5	1	3	3.46
gd30	5	0	0	0.17
gd60	9	0	0	0.5
gd91	6	0	0	0.77
gd96	11	1	4	4.89
gd83	12	1	4	4.06
gb64	15	0	1	1.21
gd26	10	1	4	4.33
gd81	10	0	0	0.61
gb73	15	2	7	7.58
gd28	8	1	4	4.07
gd67	10	3	10	10.77
gb01	11	0	0	0.66
gd82	13	3	10	10.76
gb07	10	1	4	4.29
gb03	16	1	5	5
gb08	17	1	4	4.45

A.5.5 Data for Figure 85

Investigation Parameters: Table 16

Number of Library Indicators	Total High Resolution IRM Time (s)
20	89.83
19	89.36
18	89.3
17	89.26
16	89.2
15	88.93
14	89.02
13	89.69
12	89.03
11	88.48
10	88.1
9	87.71
8	87.67
7	87.43
6	87.5
5	87.11
4	87.73
3	87.4
2	86.29
1	86.73
0	85.74

A.5.6 Data for Figure 86

Investigation Parameters: Table 17

Number of Indicates Relationships	High Resolution Total IRM Time (s)
23	97.22
29	97.71
35	97.72
41	97.98
47	98.58
53	98.53
59	98.86
65	100.25
71	100.07
77	100.51

A.5.7 Data for Figure 88, and Figure 89

Investigation Parameters: Table 18

A.5.7.1 Specialisations

Number of Specialisations	High Resolution Conclusion Generation Time (s)
1	4.77
2	5.53
3	6.42
4	7.42
5	8.29
6	9.35
7	10.12
8	10.53
9	11.04
10	11.42
11	11.87
12	12.4
13	12.79

A.5.7.2 Composites

Number of Composites	High Resolution Conclusion Generation Time (s)
4	4.73
5	4.92
6	5
7	5.22
8	5.38
9	5.38
10	5.88
11	5.93
12	6
13	6.31
14	6.58
15	6.92
16	7.04

A.6 Data for Section 8.7: Spatial Cost

Investigation Parameters: Table 18

A.6.1 Data for Figure 90, and Figure 91

A.6.1.1 Specialisations

Number of Specialisations	Number of Conclusions
1	55
2	71
3	87
4	103
5	119
6	135
7	151
8	158
9	165
10	172
11	179
12	186
13	193

A.6.1.2 Composites

Number of Composites	Number of Conclusions
4	55
5	58
6	61
7	64
8	67
9	70
10	73
11	76
12	79
13	85
14	91
15	97
16	103

A.7 Data for Section 8.8: Expandability

Investigation Parameters: Table 19

A.7.1 Data for Figure 92

A.7.1.1 Program 1

	Identifier	Keyword	Comment	Identifier + Keyword	Identifier + Comment	Keyword + Comment	Identifier + Keyword + Comment
Concepts Found	5	0	1	6	6	2	6
Percentage of Total Concepts	83	0	17	100	100	33	100

A.7.1.2 Program 2

	Identifier	Keyword	Comment	Identifier + Keyword	Identifier + Comment	Keyword + Comment	Identifier + Keyword + Comment
Concepts Found	1	0	0	3	1	0	3
Percentage of Total Concepts	33	0	0	100	33	0	100

A.8 Data for Section 8.10: Domain Independence

Investigation Parameters: Table 20

A.8.1 Data for Table 21

Program	Length (lines)	Total Concepts	Accurate Concepts	Strictly Accurate Concepts	Number of SOMs Used	Percentage Accuracy	Percentage Strict Accuracy
pn29	205	6	5	5	1	83	83
pn28	209	1	0	0	0	0	0
pk35/6	360	2	0	0	1	0	0
pn23	551	9	6	4	1	67	44
pi43	709	14	9	7	3	64	50
pi49	1104	18	6	5	4	33	28
pi41	4805	30	14	4	8	47	13
pk352prc	8076	20	12	5	4	60	25

Median Accuracy	54	Mean Accuracy	44	Standard Deviation (σ)	29
Median Strict Accuracy	27	Mean Strict Accuracy	30	Standard Deviation (σ)	26

A.9 Program Sets

A.9.1 Program Set A

Program	Length (lines)	Program	Length (lines)	Program	Length (lines)
gd95	89	gd83	547	gd82	1105
gb92/6	190	gb64	596	gd02	1117
gd25	238	gd26	650	gb07	1162
gd12	285	gd81	701	gb03	1237
gd30	337	gb73	728	gbcm0133	1310
gd60	387	gd28	807	gb08	1374
gd91	441	gd67	879		
gd96	491	gb01	1013		

A.9.2 Program Set B

Program	Length (lines)
gd95	89
gd25	238
gd12	285
gd83	547
gd28	807
gb01	1013
gd02	1117
gb08	1374

A.9.3 Program Set C

Program	Length (lines)
gd30	337
gd96	491
gd26	650
gb73	728
gd67	879
gd82	1105
gb03	1237

A.9.4 Program Set D

Program	Length (lines)
gd12	285
gd96	491
gd83	547
gd26	650
gb73	728
gd28	807
gd67	879

A.9.5 Program Set E

Program	Length (lines)	Program	Length (lines)	Program	Length (lines)
gd95	89	gd96	491	gd67	879
gb92/6	190	gd83	547	gb01	1013
gd25	238	gb64	596	gd82	1105
gd12	285	gd26	650	gb07	1162
gd30	337	gd81	701	gb03	1237
gd60	387	gb73	728	gb08	1374
gd91	441	gd28	807		

A.9.6 Program Set F

Program	Length (lines)
gb08	1374

A.9.7 Program Set G

Program	Length (lines)
gd95	89
gd25	238

A.9.8 Program Set H

Program	Length (lines)
pn29	205
pn28	209
pk35/6	360
pn23	551
pi43	709
pi49	1104
pi41	4805
pk352prc	8076

References

[BANK93] R.D. Banker, S.M. Datar, C.F. Kemerer, D. Zweig, "Software Complexity and Maintenance Costs", *Communications of the ACM,* Vol. 36, No. 11, November 1993, pp. 81-94.

[BEAL92] R. Beale, T. Jackson, *Neural Computing: An Introduction,* IOP Publishing Ltd., 1992, ISBN 0852742622.

[BIGG89] T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse", *IEEE Computer,* Vol. 22, No. 7, July 1989, pp. 36-49.

[BIGG93] T.J. Biggerstaff, B. Mitbander, D. Webster, "The Concept Assignment Problem in Program Understanding", *Proceedings of the Fifteenth International Conference on Software Engineering, Baltimore, Maryland, May 17-21, 1993,* IEEE Computer Society Press, 1993, pp. 482-498.

[BIGG94] T.J. Biggerstaff, B.G. Mitbander, D.E. Webster, "Program Understanding and the Concept Assignment Problem", *Communications of the ACM,* Vol. 37, No. 5, May 1994, pp. 72-82.

[BROO83] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies,* Vol. 18, 1983, pp. 543-554.

[BURD99] E. Burd, M. Munro, "Evaluating the Use of Dominance Trees for C and COBOL", *Proceedings of the International Conference on Software Maintenance, Oxford, England, August 30-September 3, 1999,* IEEE Computer Society Press, 1999, ISBN 0769500161, pp. 401-410.

[CHIN95] D.N. Chin, A. Quilici, "DECODE: A Cooperative Program Understanding Environment", *Journal of Software Maintenance,* Vol. 8, No. 1, 1996, pp. 3-34.

[CORB89] T.A. Corbi, "Program Understanding: Challenge for the 1990s", *IBM Systems Journal,* Vol. 28, No. 2, 1989, pp. 294-306.

[DEBA94] J-M. Debaud, B. Moopen, S. Rugaber, "Domain Analysis and Reverse Engineering", *Proceedings of the International Conference on Software Maintenance, Victoria, BC, Canada, September 19-23, 1994,* IEEE Computer Society Press, 1994, pp. 326-335.

[DESJ00] M. desJardins, P. Rheingans, "Visualisation of High-Dimensional Model Characteristics", *Proceedings of New Paradigms in Information Visualisation, Eighth ACM International Conference on Information and Knowledge Management, Kansas City, MO, USA, November 1999,* ACM Press, 2000, pp. 6-9.

[DEVA91] P. Devanbu, R.J. Brachman, P.G. Selfridge, B.W. Ballard, "LaSSIE: A Knowledge-Based Software Information System", *Communications of the ACM,* Vol. 34, No. 5, May 1991, pp. 35-49.

[EMAM98] K.E. Emam, J-N. Drouin, W. Melo, *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination,* IEEE Computer Society, 1998, ISBN 0818677988.

[FJEL79] R.K. Fjeldstad, W.T. Hamlen, "Application Program Maintenance Study - Report to Our Respondents", in [PARI83], 1979, pp. 13-27.

[GALL91] K.B. Gallagher, J.R. Lyle, "Using Program Slicing In Software Maintenance", *IEEE Transactions on Software Engineering,* Vol. 17, No. 8, August 1991, pp. 751-761.

[GELL91a] E.M. Gellenbeck, C.R. Cook, "An Investigation of Procedure and Variable Names as Beacons during Program Comprehension", in *Empirical Studies of Programmers: Fourth Workshop, New Brunswick, NJ, December 7-9, 1991,* J. Koenemann-Belliveau, T.G. Moher, S.P. Robertson (editors), Ablex Publishing Corporation, Norwood, New Jersey, 1991, ISBN 0893918571, pp. 65-81.

[GELL91b] E.M. Gellenbeck, C.R. Cook, "Does Signalling Help Professional Programmers Read And Understand Computer Programs?", in *Empirical Studies of Programmers: Fourth Workshop, New Brunswick, NJ, December 7-9, 1991,* J. Koenemann-Belliveau, T.G. Moher, S.P. Robertson (editors), Ablex Publishing Corporation, Norwood, New Jersey, 1991, ISBN 0893918571, 1991, pp. 82-98.

[HALL87a] R.P. Hall, "Seven Ways to Cut Software Maintenance Costs (Digest)", in *Techniques of Program & System Maintenance, 2nd edition,* G. Parikh, QED Information Sciences Inc., 1988, pp. 358-360.

[HALL87b] R.P. Hall, "Seven Ways to Cut Software Maintenance Costs", *Datamation,* Vol. 33, July 15, 1987, pp. 81-84.

[HAME96] L.G.C. Hamey, J.C.-H. Yeh, "Segmentation of Bake Images by a Self-Organising Map", *Image Segmentation Workshop,* Australian Pattern Recognition Society, 1996, pp. 65-68.

[HARA90] M.T. Harandi, J.Q. Ning, "Knowledge-Based Program Analysis", *IEEE Software,* Vol. 7, No. 1, January 1990, pp. 74-81.

[HART91a] J. Hartman, "Automatic Control Understanding for Natural Programs", *Ph.D. Thesis,* University of Texas at Austin, May 1991.

[HART91b] J. Hartman, "Understanding Natural Programs using Proper Decomposition", *Proceedings of the Thirteenth International Conference on Software Engineering, Austin, Texas, May 13-17, 1991,* IEEE Computer Society/ACM Press, 1991, pp. 62-73.

[HART92] J. Hartman, "Pragmatic, Empirical Program Understanding", *Workshop Notes, First Workshop on Artificial Intelligence & Automatic Program Understanding, Tenth National Conference on Artificial Intelligence, San Jose, California,* July 12-16, 1992.

[HAZA93] J.E. Hazan, S.A. Jarvis, R.G. Morgan, R. Garigliano, "Understanding Lolita: Program Comprehension in Functional Languages", *Proceedings of the Second IEEE Workshop on Program Comprehension, Capri, Italy, July 1993,* IEEE Computer Society Press, 1993, pp. 26-34.

[HONK97] T. Honkela, "Self-Organising Maps in Natural Language Processing", *Ph.D. Thesis,* Helsinki University of Technology, 1997.

[IEEE98] IEEE Computer Society, *IEEE Standard for Software Maintenance (IEEE Std 1219-1998),* Institute of Electrical and Electronics Engineers, 1998, ISBN 0738103365, in *IEEE Standards, Software Engineering Volume 2 Process Standards, 1999 Edition,* Institute of Electrical and Electronics Engineers, 1999, ISBN 0738115606.

[ISO99] International Standards Organisation, *International Standard: Information Technology - Software Maintenance (ISO/IEC 14764:1999),* International Standards Organisation, 15th November 1999.

[JOHN85] W.L. Johnson, E. Soloway, "PROUST: Knowledge-Based Program Understanding", *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 3, March 1985, pp. 267-275.

[JOHN86] W.L. Johnson, *Intention-Based Diagnosis of Novice Programming Errors*, Morgan Kaufmann Publishers Ltd, 1986, ISBN 0273087681.

[KARA92] V. Karakostas, "Intelligent Search and Acquisition of Business Knowledge from Programs", *Software Maintenance: Research and Practice*, Vol. 4, 1992, pp. 1-17.

[KASK96] S. Kaski, T. Honkela, K. Lagus, T. Kohonen, "Creating an Order in Digital Libraries with Self-Organising Maps", *Proceedings of WCNN'96, World Congress on Neural Networks, San Diego, California, September 15-18, 1996*, Lawrence Erlbaum and INNS Press, Mahwah, NJ, 1996, pp. 814-817.

[KOHO96] T. Kohonen, J. Hynninen, J. Kangas, J. Laaksonen, "SOM_PAK: The Self-Organizing Map Program Package", *Technical Report A31, Laboratory of Computer & Information Science, Helsinki University of Technology*, ISBN 9512229471, January 1996.

[KOHO97] T. Kohonen, *Self-Organizing Maps, Second Edition*, Springer, 1997, ISBN 3540620176.

[KOHO00] T. Kohonen, *The Self-Organizing Map (SOM)*, http://www.cis.hut.fi/projects/somtoolbox/somintro/som.html, 2000.

[KOZA94] W. Kozaczynski, J.Q. Ning, "Automated Program Understanding By Concept Recognition", *Automated Software Engineering*, Vol. 1, No. 1, March 1994, pp. 61-78.

[KUWA97] Y. Kuwata, M. Yatsu, "Managing Knowledge Using A Semantic Network", *Proceedings of AAAI Spring Symposium on Artificial Intelligence in Knowledge Management, Stanford University, March 24-46, 1997*, B.R. Gaines, R. Uthurusamy (co-chairs), Technical Report SS-97-01, AAAI Press, 1997, ISBN 1577350243, pp. 94-98.

[LAGU96] K. Lagus, T. Honkela, S. Kaski, T. Kohonen, "Self-Organizing Maps of Document Collections: A New Approach to Interactive Exploration", *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, E. Simoudis, J. Han, U. Fayyad (editors), AAAI Press, Menlo Park, California, 1996, pp. 238-243.

[LIEN80] B.P. Lientz, E.B. Swanson, *Software Maintenance Management*, Addison-Wesley Publishing Company, 1980, ISBN 0201042053.

[LITT86] D.C. Littman, J. Pinto, S. Letovsky, E. Soloway, "Mental Models and Software Maintenance", in *Empirical Studies of Programmers: First Workshop, June 5-6, 1986, Washington, DC*, E. Soloway, S. Iyengar (editors), Ablex Publishing Corporation, Norwood, New Jersey, 1987 (second printing), ISBN 0893914630, pp. 80-98.

[MAYR94] A. von Mayrhauser, A.M. Vans, "Comprehension Processes During Large Scale Maintenance", *Proceedings of the Sixteenth International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*, IEEE Computer Society/ACM Press, 1994, pp. 39-48.

[MAYR95] A. von Mayrhauser, A.M. Vans, "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer*, Vol. 28, No. 8, August 1995, pp. 44-55.

[MAYR97] A. von Mayrhauser, A.M. Vans, A.E. Howe, "Program Understanding Behaviour During Enhancement of Large-scale Software", *Software Maintenance: Research and Practice*, Vol. 9, No. 5, 1997, pp. 299-327.

[MAYR98] A. von Mayrhauser, A.M. Vans, "Program Understanding During Software Adaptation Tasks", *Proceedings of the International Conference on Software Maintenance, Bethesda, Maryland, November 16-19, 1998*, IEEE Computer Society Press, 1998, pp. 316-325.

[MERK97] D. Merkl, "Lessons Learned in Text Document Classification", *Workshop on Self-Organizing Maps (WSOM'97), Helsinki, Finland, June 4-6, 1997,* pp. 316-321.

[MIAR83] R. J. Miara, J.A. Musselman, J.A. Navarro, B. Shneiderman, "Program Indentation and Comprehensibility", *Communications of the ACM,* Vol. 26, No. 11, November 1983, pp. 861-867.

[NEUR00] T*he Neural Network FAQ, Part 1 of 7,* ftp://ftp.sas.com/pub/neural/FAQ.html.

[NING94] J.Q. Ning, A. Engberts, W. Kozaczynski, "Automated Support For Legacy Code Understanding", *Communications of the ACM,* Vol. 37, No. 5, May 1994, pp. 50-57.

[PARI83] G. Parikh, N. Zvegintzov (editors), *Tutorial on Software Maintenance,* IEEE Computer Society, 1983, ISBN 0818600020.

[PAUL93] M.C. Paulk, B. Curtis, M.B. Chrissis, C.V. Weber, "Capability Maturity Model for Software, Version 1.1", *Technical Report CMU/SEI-93-TR-024, ESC-TR-93-177,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, February 1993.

[PENN87] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs", *Cognitive Psychology,* Vol. 19, No. 2, February 1987, pp. 295-341.

[QUIL96] A. Quilici, S. Woods, "Toward a Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms", *Proceedings of the Fourth International Workshop on Program Comprehension, Berlin, Germany, March 29-31, 1996,* IEEE Computer Society Press, March 1996, pp. 55-64.

[QUIL97] A. Quilici, S. Woods, Y. Zhang, "New Experiments with a Constraint-Based Approach to Program Plan Matching", *Proceedings of the Fourth Working Conference on Reverse Engineering, Amsterdam, The Netherlands, October 6-8, 1997,* I. Baxter, A. Quilici, C. Verhoef (editors), IEEE Computer Society Press, 1997, pp. 114-123.

[QUIL98] A. Quilici, Q. Yang, S. Woods, "Applying Plan Recognition Algorithms To Program Understanding", *Automated Software Engineering,* Vol. 5, No. 3, July 1998, pp. 1-26.

[RAMA96] S. Ramanujan, "An Experimental Investigation of the Impact of Individual, Program and Organisational Characteristics on Software Maintenance Effort", *Proceedings of the Second Americas Conference on Information Systems, Phoenix, Arizona,* J.M. Carey (editor), August 16-18, 1996, Association for Information Systems. http://hsb.baylor.edu/ramsower/ais.ac.96/papers/ramanujan.htm.

[RICH88] C. Rich, R.C. Waters, "The Programmer's Apprentice: A Research Overview", *IEEE Computer,* Vol. 21, No. 11, November 1988, pp. 10-25.

[RICH90] C. Rich, R.C. Waters, *The Programmer's Apprentice,* ACM Press (Frontier Series), 1990, ISBN 0201524252.

[RICH92] C. Rich, R.C. Waters, "Knowledge Intensive Software Engineering Tools", *IEEE Transactions on Knowledge and Data Engineering,* Vol. 4, No. 5, October 1992, pp. 424-430.

[RICH93] C. Rich, R.C. Waters, "Approaches to Automatic Programming", *Advances in Computers,* Vol. 37, 1993, pp. 1-57.

[ROBS91] D.J. Robson, K.H. Bennett, B.J. Cornelius, M. Munro, "Approaches to Program Comprehension", *Journal of Systems and Software,* Vol. 14, February 1991, pp. 79-84.

[ROME99] P. Romero, "Focal Structures in Prolog", *Proceedings of the Psychology of Programming Interest Group (PPIG), Eleventh Workshop, University of Leeds, UK, January 5-7, 1999.*
http://www.ppig.org/papers/11th-romero.pdf

[ROUS98] D. Roussinov, H. Chen, "A Scalable Self-Organising Map Algorithm for Textual Classification: A Neural Network Approach to Thesaurus Generation", *Communication and Cognition - Artificial Intelligence,* Vol. 15, No. 1-2, 1998, pp. 81-112.

[RUGA96] S. Rugaber, K. Stirewalt, L.M. Wills, "Understanding Interleaved Code", *Automated Software Engineering,* Vol. 3, No. 1/2, July 1996, pp. 47-76.

[SAYY97] J. Sayyad-Shirabad, T.C. Lethbridge, S. Lyon, "A Little Knowledge Can Go A Long Way Towards Program Understanding", *Proceedings of the Fifth International Workshop on Program Comprehension, Dearborn, Michigan, May 28-30, 1997,* IEEE Computer Society Press, pp. 111-117.

[SELF93] P.G. Selfridge, R.C. Waters, E.J. Chikofsky, "Challenges to the Field of Reverse Engineering", *Proceedings of the Working Conference on Reverse Engineering, Baltimore, Maryland, May 21-23, 1993,* IEEE Computer Society Press, 1993, pp. 144-150.

[SOLO84] E. Soloway, K. Ehrlich, "Empirical Studies of Programming Knowledge", *IEEE Transaction on Software Engineering,* Vol. SE-10, No. 5, September 1984, pp. 595-609.

[SOLO88] E. Soloway, B. Adelson, K. Ehrlich, "Knowledge and Processes in the Comprehension of Computer Programs", in *The Nature Of Expertise,* M.T.H. Chi, R. Glaser, M.J. Farr (editors), Lawrence Erlbaum Associates, 1988, ISBN 0805804048, pp. 129-152.

[SOMM93] I. Sommerville, *Software Engineering*, Addison-Wesley Publishers Ltd, USA, 1993, ISBN 0201565293.

[SOMP00] *SOM_PAK: The Self-Organizing Map Program Package*, http://www.cis.hut.fi/research/som-research/nnrc-programs.shtml

[SOMT00] *SOM Toolbox for Matlab*, http://www.cis.hut.fi/projects/somtoolbox/somalg.shtml.

[STAN84] T.A. Standish, "An Essay on Software Reuse", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 494-497.

[STOR97] M-A.D. Storey, F.D. Fracchia, H.A. Müller, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization", *Proceedings of the Fifth International Workshop on Program Comprehension, Dearborn, Michigan, May 28-30, 1997*, IEEE Computer Society Press, 1997, pp. 17-28.

[STOR98] M-A.D. Storey, "A Cognitive Framework for Describing and Evaluating Software Exploration Tools", *Ph.D. Thesis*, Simon Fraser University, December 1998.

[SWAN76] E.B. Swanson, "The Dimensions of Maintenance", *Proceedings of the Second International Conference on Software Engineering, San Francisco, October 13-15*, 1976, pp. 492-297.

[TEAS94] B.E. Teasley, "The Effects of Naming Style and Expertise on Program Comprehension", *International Journal of Human Computer Studies*, Vol. 40, 1994, pp. 757-770.

[TILL95] S. Tilley, D.B. Smith, *Perspectives on Legacy System Reengineering*, http://www.sei.cmu.edu/reengineering/pubs/lsysree/lsysree.html.

[TILL96a] S.R. Tilley, S. Paul, D.B. Smith, "Towards a Framework for Program Understanding", *Proceedings of the Fourth International Workshop on Program Comprehension, Berlin, Germany, March 29-31, 1996,* IEEE Computer Society Press, March 1996.

[TILL96b] S.R. Tilley, "Coming Attractions in Program Understanding", *Technical Report CMU/SEI-96-TR-019, ESC-TR-96-019,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, December 1996.

[TILL98a] S.R. Tilley, "Coming Attractions in Program Understanding II: Highlights of 1997 and Opportunities in 1998", *Technical Report CMU/SEI-98-TR-001, ESC-TR-98-001,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, February 1998.

[TILL98b] S.R. Tilley, "A Reverse-Engineering Environment Framework", *Technical Report CMU/SEI-98-TR-005, ESC-TR-98-005,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, April 1998.

[TIP94] F. Tip, "A Survey of Program Slicing Techniques", *Technical Report CS-R9438,* Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[VESA97] J. Vesanto, "Using the SOM and Local Models in Time-Series Prediction", *Proceedings of Workshop on Self-Organizing Maps (WSOM97), Helsinki, Finland, June 4-6, 1997.*

[WATE79] R.C. Waters, "A Method for Analyzing Loop Programs", *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 3, May 1979, pp. 237-250.

[WATE82] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 1, January 1982, pp. 1-12.

[WIED86] S. Wiedenbeck, "Beacons In Computer Program Comprehension", *International Journal of Man-Machine Studies*, Vol. 25, 1986, pp. 697-709.

[WIED91] S. Wiedenbeck, "The Initial Stage of Program Comprehension", *International Journal of Man-Machine Studies*, Vol. 35, 1991, pp. 517-540.

[WILL90] L.M. Wills, "Automated Program Recognition: A Feasibility Demonstration", *Artificial Intelligence*, Vol. 45, No. 1-2, September 1990, pp. 113-172.

[WILL92] L.M. Wills, "Automated Program Recognition by Graph Parsing", *PhD Thesis*, AI Lab, Massachusetts Institute of Technology, July 1992.

[WILL93] L.M. Wills, "Flexible Control for Program Recognition", *Proceedings of the Working Conference on Reverse Engineering, Baltimore, Maryland, May 21-23, 1993*, IEEE Computer Society Press, 1993, pp. 134-143.

[WOOD96a] S. Woods, A. Quilici, "Some Experiments Toward Understanding How Program Plan Recognition Algorithms Scale", *Proceedings of the Third Working Conference on Reverse Engineering, Monterey, California, November 8-10, 1996*, L. Wills, I. Baxter, E. Chikofsky (editors), IEEE Computer Society Press, 1996, pp. 21-30.

[WOOD96b] S. Woods, Q. Yang, "The Program Understanding Problem: Analysis and a Heuristic Approach", *Proceedings of the Eighteenth International Conference on Software Engineering, Berlin, Germany, March 25-30, 1996*, IEEE Computer Society Press, 1996, pp. 6-15.

[WOOD98a] S. Woods, Q. Yang, "Program Understanding as Constraint Satisfaction: Representation and Reasoning Techniques", *Automated Software Engineering,* Vol. 5, No. 2, April 1998, pp. 147-181.

[WOOD98b] S.G. Woods, A.E. Quilici, Q. Yang, *Constraint-Based Design Recovery for Software Reengineering: Theory and Experiments,* Kluwer Academic Publishers, 1998, ISBN 0792380673.

[ZHAN97] Y. Zhang, "Scalability Experiments in Applying Constraint-Based Program Understanding Algorithms to Real-World Programs", *M.Sc. Thesis,* University of Hawaii, May 1997.